Department of Computer Science

Technical Report

Undergraduate Dissertation: Csound Parallelism

Christopher Wilson

Technical Report 2009-07 **May 2009**

ISSN 1740-9497
Editor: Dr Marina De Vos

Contact Address:
Technical Report Editor
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497
Editor: Dr Marina De Vos

Csound Parallelism

Christopher Wilson

Bachelor of Science in Computer Science with Honours
The University of Bath
April 2009

Csound Parallelism

Submitted by: Christopher Wilson

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

Recent developments in processor technology have focused on multiple processors ahead of increases in clock speed. To take advantage of this serial programs like Csound need modification to split out the parallel parts. We investigate other parallel programming languages which bear some resemblance to Csound. We consider a combination of the approach taken by Fitch and Marti (1980s) and ideas from instruction scheduling to be a good basis for our work. We performed several iterations on designs based upon these ideas, finishing with a design that gives speed improvements over a range of tests. We note that there is still work to do on machines with more than two processors and to fix some small semantic issues caused by reordering of floating point operations. We conclude that it is necessary to take into account the ratio of gains from parallelism versus the parallelism overhead. We also have to consider the scheduler of the underlying operating system and design our program to take advantage of it.

Contents

List of Figures

List of Tables

Acknowledgements

We would like to acknowledge our supervisor for his support and guidance throughout the project. We would also like to acknowledge Zaid Al-Chalabi for allowing us the use of his machine to test on.

Chapter 1

Introduction

1.1 Csound

> Csound is a sound design, music synthesis, and signal processing system, providing facilities for composition and performance over a wide range of platforms. (`http://www.csounds.com`)

The Csound program is an interpreter which accepts as input a pair of programs in the following languages:

Orchestra a definition of the instruments that will play.

Score a definition of when and what the instruments should play.

These can be combined in a markup language inspired format called `csd` or as two separate files. The choice does not effect the output so we will just refer to inputs in the two languages. Csound can output your created sound to a file or in real-time to the audio output of your computer. Orchestras can be reused by many score programs and in theory the reverse is also true, but not very common.

A description of the internals of Csound `v4` is available at `http://www.csounds.com/internals`. Summarised: the output for each sample is calculated from the instruments scheduled to be playing at that time. This is calculated sequentially, enabling communication between multiple instruments through global variables with a (predictable) serial ordering.

1.2 Parallelism

Why are we parallelising anything? It has been suggested that Moore's law will end, either because transistors cannot shrink forever or because of power and heat issues (Wolfe, 2004).

Processor manufacturers have turned to multicore[1] processors as a means to extend Moore's law (Geer, 2005). It is not just super computers that are effected, consumer electronics such as laptops and game consoles are included in this transition. Examples include the Xbox 360 (Andrews and Baker, 2006) and the Playstation 3 (Hofstee, 2005).

Serial programs have one stream of instructions, that one processor can execute. To use the other processors we need to split out the parallel parts of our program so we can provide other streams of instructions for execution on the other processors. If we split up the program so that independent work is done in parallel we should finish faster. If we run just our serial program, the operating system is left to find a use for the non-utilised processors.

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Figure 1.1: Flynn's Taxonomy

The basic architectures for parallel computers are shown in Figure 1.1 (Flynn, 1972). Within the MIMD group there is a division based on the memory architecture.

Shared memory can be accessed by any of the processors in the machine.

Distributed memory, each processor has its own block of memory. For a processor to access data stored in the memory of another, it must ask for it.

Most consumer processors fall into the MIMD shared memory category. They feature some support for SIMD in the form of SSE (Raman, Pentkovski and Keshava, 2000) or AltiVec (Seebach, 2005)—for x86 and PowerPC respectively—instruction sets that operate on vectors.

The basic models for parallelism are:

bit is the increase in the size of the word operated on by instructions. Adding 32-bit values in a 32-bit machine requires one step. It requires more in a 16-bit machine, and even more in an 8-bit machine.

instruction is the reordering of instructions and pipelining that happen in modern processors and their associated compilers (Rau and Fisher, 1993).

data is the concurrent application of a function to a dataset which is spread across multiple processors (Lewis, 1997). For example if we were to add two n-dimensional vectors we could put $1–\frac{n}{2}$ elements of each vector on one processor and $\frac{n}{2}–n$ on another.

[1] multicore is multiple CPUs on the same chip

Each processor can then add the corresponding components, at the end we join the vector back together to form the result. Because the operation on any element is independent of the others we can spread the vector out to as many processors as necessary.

task is the evaluation of multiple tasks concurrently. For example we have a function which interprets an array and produces data to plot on a graph. We could compute the function on one array on one processor, compute again with a different array on another. Then plot the results of both on the same graph.

We will refer to cores as processors throughout the rest of this document. For our purposes a dual core processor machine and a dual processor machine are both MIMD machines.

1.3 Aims

So we need to implement parallelism on a consumer multi-processor machine, which means targeting a MIMD architecture. **bit** and **instruction** parallelism are already decided by our machine architecture or handled by our compiler. That leaves `data` and `task` parallelism for us to work with. Our parallel implementation should give the same output as the serial version and should be faster on a consumer multi-processor machine.

This Document

We have the aims for our project, in Section 2 we research prior work in this area to guide our design decisions. In Section 3 we draw upon the previous work and our aims to produce a set of requirements for the project. Section 4 discusses how we will test our work. Section 5 is a detailed description of the design and implementation of the project. Finally Section 6 draws conclusions, reflects and suggests future work.

Chapter 2

Literature Review

We first discuss some finer details of Csound in Section 2.1. Then, who tells Csound to execute a task in parallel and how is it done? How much faster can it make Csound? In Section 2.2 we will decide who parallelises the program. In Section 2.3 we will examine the potential speed gains. The next two sections (Sections 2.4 and 2.5) will visit some other parallel programming languages to see how they implemented parallelism. Section 2.6 will build on this to examine how we might decide what to run in parallel. Section 2.7 will feature our choice of how to implement parallelism. Finally Section 2.8 will present the closest implementation to our expected target.

2.1 Csound Details

We are building upon the current 5.10.1 release of Csound. We first take a closer look at Csound so we know what can be done in parallel. We mentioned in the Introduction that there are instruments defined in orchestra files and times for the instruments to play defined in score files.

Multiple instruments can be playing at the same time, including multiple versions of the same instrument. There is a level of abstraction introduced by the score program, so we can determine what instruments are playing at a certain point in time, but it is not immediately obvious.

Variables in Csound come in four kinds:

setup-only
i-rate scalar value that changes with the starting / stopping of instruments.
k-rate scalar value that is updated at the `k`-rate.
a-rate vector value that is also updated at the `k`-rate.

The `k`-rate is specified by the programmer in the orchestra. For example the sample-rate (`a`-rate) may be 44100 and the `k`-rate 4410. So 4410 times for every second of the piece

we are computing we: determine what instruments are playing, update the `k`-variables, perform each opcode, calculating the a-rate vectors along the way. In our example the size of the a-rate vector will be 10 samples.

There is also the further dichotomy of local and global variable. Local variables come with the prefixes `p`, `i`, `k` and `a` in the order outlined above above. Global variables are a reserved symbol or have a `g` prefix in front of the local prefix.

The Csound manual contains a detailed description of the various kinds of variables[1].

The most common use of global variables in Csound is to add a reverberation (reverb) effect to a piece of music. As instruments with higher numbers play later a reverb opcode can be put in the highest numbered instrument and left on for the duration that other instruments are playing. When other instruments write to the output they also write their output into a global variable. The reverb instrument passes that global variable to the reverb opcode and finally writes the result to the output.

2.2 Automatic or Manual?

We need to add parallel constructs to Csound to tell it that it should compute something concurrently. Where we put these constructs defines the type of parallelism.

Manual Constructs are added to the input languages: orchestra and/or score. There is a corresponding underlying implementation in Csound that is run whenever the programmer makes the appropriate call in the input program.

Automatic Constructs are added to just Csound. Csound examines the input program and determines which parts can be computed in parallel. There are no changes to the languages.

The users of Csound are musicians who happen to be writing a program to make music. They may or may not be trained programmers. It follows that they should not really have to care whether the output of their instruments is being calculated at the same time. It is not currently and they are happy with the results. So we will be using automatic parallelisation. There is also evidence that producing a parallel version of an existing serial program is significantly more difficult than the original serial version (Hochstein, Carver, Shull, Asgari and Basili, 2005).

We also mentioned in Section 2.1 that when writing an orchestra the programmer does not know how many instruments will be playing. When writing the score the programmer does not know which instruments are computationally heavy. This makes it difficult to add constructs to any one of the languages in a similar fashion to Pthreads or MPI. A programmer could split up work in one instrument in the orchestra, then find that eight of those instruments are playing when the machine only has 2 processors.

[1] `http://www.csounds.com/manual/html/OrchKvar.html`

Automatic parallelisation has the addition benefit that no changes will be required to existing source code for it to run in parallel. There is a substantial body of Csound programs already existing which would require an enormous amount of effort to convert. We will not be altering the Csound languages.

2.3 Amdahl's Law and our Goals

To generate a reasonable goal for the expected speedup for our program we turn to Amdahl's Law (Equation 2.1) (Amdahl, 2000). This states that the expected speedup of the parallel version of a program is proportional to the serial component. i.e. the serial component has to be very small to achieve a significant speedup. Throwing additional processors at a problem will have diminishing returns.

where:

$s =$ serial proportion of program
$p =$ parallel proportion
$n =$ number of processors

$$Speedup = \frac{1}{s + \frac{p}{n}}. \tag{2.1}$$

Further enquiry finds that for some situations the p and n components are related (Equation 2.2) (Gustafson, 1988). That is in practise having more processors allows you to tackle a larger problem in the same time.

$$Speedup = n + (1 - n) * s. \tag{2.2}$$

So taking an input Csound program that is already computable in real time and expecting some sort of improvement is unreasonable. However if we focus on programs that are not computable in real time and attempt to make them so, we will have made real gains. Of course any improvements to the non-realtime rendering time of larger pieces will be a benefit.

2.4 A Data Parallel Language—High Performance Fortran

High Performance Fortran (HPF) is a data parallel programming language. It is an extension to the Fortran 90 standard and builds on the array operations introduced therein (Loveman, 1993)[2].

[2]As a historic note some of the features implemented in HPF were adopted in the Fortran 95 standard.

The array operations provided the programmer with a means of performing operations—including functions—on whole arrays. This is in general a closer match to the specification a scientific programmer will be working from. An example is shown with the old style in Listing 2.1 and the new style in Listing 2.2. The general concept is similar to that of using map in a functional language.

Listing 2.1: Create an Array and set all values to 1

```
1 real a(20)
2 do i = 1,len(20)
3       a(i) = 1.0
4 end do
```

Listing 2.2: Create an Array and set all values to 1

```
1 real a(20) = 1.0;
```

HPF provides a means to annotate the source code with distribution directives for elements of the arrays. The programmer specifies how to lay the data out across abstract processors and the runtime lays out the data across real processors using this information. The array operations can then be executed in parallel, each processor operating on its data.

The FORALL statement (HPF) is used to specify that everything in the loop is independent of the iteration number i.e. different iterations can be executed on different processors. The PURE statement is added to a function if it contains no side effects, only PURE functions can be called in a FORALL loop.

Before we continue we need to consider the semantics of some constructs in the HPF language. Some constructs like array operations define not just the operation but also mutual exclusion. This is interesting to us as we can see they have inferred additional information from an existing language without requiring the programmer to specify it. This is the kind of information we require to implement automatic parallelism. Alternately the distribution directives serve only one purpose: to inform the compiler how the parallel operations should be performed. Unfortunately the array operations require the distribution directives to compute in parallel.

There is no parallelism of this kind available to us in the Csound languages. However it is worth exploring as we may find that some of the internals of the Csound interpreter could benefit from this. With appropriate level of compiler optimisation the `out` family of opcodes may already use vector instructions if the processor supports them.

2.5 Some Task Parallel Programming Languages

We are now going to investigate some previous programming languages. While none of them are fully automatic we will examine them for situations similar to the array operations in

HFP (Section 2.4). That is where additional semantics relevant to parallel computation are extracted from existing language constructs. We will also look at how they split up the available work between processors.

2.5.1 Fx

Fx is an based on a mixture of FORTRAN 77, Fortran 90 and HPF (Subhlok, Stichnoth, O'Hallaron and Gross, 1993; Gross, OHallaron and Subhlok, 1994). It implements a system of array operations and layouts similar to HPF.

A mechanism for task parallelism is also included. begin parallel and end parallel mark the scope for a parallel operation. Inside this block only loops and subroutine calls are allowed. Following each subroutine call the programmer must specify the input and output parameters of that call. Further information specifying which processor to assign tasks to can also be included. When the program is compiled a task graph is generated using this information. Following this the task graph is partitioned into a module graph where each node is a module and each arc communication between them. Each module is then assigned to a processor in the real machine.

2.5.2 Assign

Assign is a programming language developed for use in the digital signal processing (DSP) domain (O'Hallaron, 1991). In DSP programs can often be represented as flowgraphs. That is as a series of functions performing some operation on inputs and producing an output. Pipelines connect the output of one function to the input of another. These pipelines are FIFOs so they can grow as necessary and provide a built in synchronisation. That is when there is no work a node will wait until there is enough input in its pipeline.

The programmer writes the functions for each node, these are then connected together by another program. At the time of the aforementioned paper the programmer wrote another program (the graph generator) that used library functions to connect the nodes.

The compiler takes the flow graph output by the graph generator program and given some options about the target machine produces an executable parallel program for that machine.

2.5.3 Gabriel

Gabriel is another language developed for use in DSP. It features many similar concepts to Assign (Section 2.5.2) (Lee, Ho, Goei, Bier and Bhattacharyya, 1989). The nodes are replaced by stars. The connection of stars is done in a graphical environment and the stars again have input and output parameters. This time the compiler checks the usage of the @[3] operator to determine how large the buffers should be for input/output.

[3] @x specifies the sample x iterations ago in input or output

The compiler also determines the order the functions should be called in and the number of times they should be called on serial or parallel versions of the program. As the input and output size is bounded we can use a program written this way for real time work.

Gabriel is implemented in Lisp, unfortunately the means of running on the parallel machine is to execute a function in each star which outputs the star as assembly. This decision was made when C compilers were not widely available for the DSP specific processors Gabriel was targeting. Gabriel is an interesting language as a comparison to Assign, but for our work there is not much to take away from it.

2.5.4 Jade

Jade is a general purpose task parallel programming language, implemented as an extension to C (Rinard, Scales and Lam, 1993). Jade uses the abstraction of a single shared memory that all tasks can access. Objects are declared statically or dynamically there with the shared keyword, and the create_object(type, size) function is used to allocate dynamic memory. The basic construct for parallelism is the withonly do construct (Listing 2.3). How and which shared variables will be used in a task are declared before the task. The access specification statements rd and wr take the shared variable as an argument and specify how it will be used. This section can build these accesses dynamically using conditionals, loops and function calls.

Listing 2.3: The withonly do construct

```
1 withonly {
2     shared variable access declaration
3     } do (parameters for task body) {
4     task body
5 }
```

When the program is run the runtime executes the access declaration to determine whether this task can execute. If it can then the task body is executed. If it uses a variable another task is using then it waits in a pool of tasks. As processors become free tasks are taken from the pool to be executed.

As the Jade runtime determines what tasks will be executed in parallel, it also determines whether there is sufficient work in the task to make it worth doing in parallel.

2.5.5 Linda

Linda is a parallelism concept which is added to other languages to create a parallel dialect of the language (Carriero and Gelernter, 1989). It uses the tuple space concept: processes communicate via a pool which data objects—called tuples—can be put into and retrieved from. New processes are created in the same way, they are put into the pool as live

tuples. When the new process finishes it turns into an ordinary data object tuple. Data structures can be built in the tuple space: either output from other tuples or in a fine grained parallelism manner. Where live tuples are released and each turns into part of the data structure when it has finished computing.

Linda provides four basic operations on the tuple space:

eval create a new process and put it in the tuple space.
out put a new data object in the tuple space.
in read in a tuple, removing it from the tuple space.
rd read in a tuple, leaving it remaining in the tuple space.

`in` and `rd` both provide the ability to use wildcards when specifying what tuple they should remove. This can be used to fetch all elements of some data structure with a known item in the tuple as they become available, by looping on the `in` operation.

The fashion in which the programmer has to write the program to use Linda exposes the available parallelism therein. Shared variables have to be put in the tuple space and can only be used by one process at a time as they read the tuple `in` and then `out` it back. Elements of the program which can occur in parallel naturally do so as the runtime selects processes which are not blocked waiting on some shared resource tuple.

Cohen was optimistic about forthcoming computers which would allow much more fine grained parallelism. That is Linda does not decide explicitly what to do in parallel, it creates new live tuples as instructed. The underlying language runtime or operating system scheduler decides where the process is executed. If a program written in the idiomatic fine grained fashion is not sufficiently fast, the programmer must alter it to move the work into fewer processes.

2.5.6 Data-Flow Languages

Data-flow languages are based on the idea of modelling a program as a set of operator nodes with edges between them delivering data (Whiting and Pascoe, 1994). Operations occur whenever all the data on the incoming edges is available, meaning there is no current operator. Once data has been sent out on an arc it cannot be changed.

There are numerous languages implementing data-flow which allow a programmer to write programs for a data-flow machine. Most of the use single assignment semantics to model the data tokens which are invariable once on an edge. We will not look at any of the specific languages here as Csound very much does have variables that vary.

The underlying data-flow architecture is interesting in a similar way to Linda: in that when writing a program we have to write it in a way which exposes the parallelism and makes the parallelism constraints obvious. A constraint is an edge from one operator node to another. Parallelism is non-connected sub-graphs: these have no edges between them and so will never be waiting for a data token from each other.

2.5.7 Conclusion

We have seen several variations on how automatic a task parallel programming language can be. Fx used the concept of a block of code where the subroutines called, could not have side effects. Their inputs and outputs were specified (a Fortran idiosyncrasy) and the compiler used these and the order to determine connections. Assign and Gabriel followed a similar concept except the programmer was responsible for connecting the inputs and outputs. The compiler then builds a task graph with the subroutines/nodes/stars as nodes and input output connections as arcs. Finally the compiler was then free to arrange the program across processors preserving the serial semantics.

Jade takes this concept a step further. Use of shared data in a block of code is declared before hand. The lexical order of the blocks is used to determine the dependencies between the shared data. The runtime is then free to execute any blocks whose conditions on mutual exclusivity are satisfied. So if two blocks are mutually exclusive they can be computed concurrently.

Linda works in a similar fashion but with much greater changes to the structure of the program. The program specifies all the parallelism constraints by means of putting and removing tuples from the tuple pool. The runtime is free to execute any processes which are not currently waiting for something in the pool.

Data-flow languages work in a similar fashion to Linda where the machine is able to compute any operation which has all its operands. We specify the parallelism constraints with the edges that link operators for data to travel over.

We can see that all the languages are concerned about usage of variables, who uses what and in what order. If we can determine this then we can determine whether the use is mutually exclusive and hence possible to compute concurrently. We can also use this to build a task graph and determine what order to do the work in to preserve the serial semantics.

2.6 Scheduling

Once we have the mutual exclusivity of parallel sections of work how do we build the task graph? When we have the task graph how do we decide what work to do on each thread?

We see that this problem of building the task graph has similar properties to instruction scheduling. Global variables are analogous to registers / memory locations and instruments to opcodes. Gibbons and Muchnick (1986) describe a method in which instructions can be reordered to minimise interlocks (in RISC machines) while preserving correctness. A dependency directed acyclic graph (DAG) on instructions is constructed by scanning backwards through the basic block noting use of resources and later uses which must precede it. The DAG is then topologically sorted to give this an equivalent—whilst still correct—order that the instructions can be performed in. The topological sort makes use of some heuristics to attempt to produce a more optimal—but not completely—order. The three

heuristics were to choose an instruction:

1. which interlocked with any of its immediate successors in the DAG.
2. with the greatest number of immediate successors.
3. with the greatest length in the path from the instruction to the leaves of the DAG.

These biased the sorting algorithm to doing those instructions which interlocked, were a requirement for a greater number of other instructions and had a large sequence of instructions following them. These options also uncovered the greatest number of instructions to be picked at the next step.

We can use these ideas for our instruments and global variables. We construct the DAG in the way described above with globals / registers and instruments / opcodes. We can perform the same scan backwards to determine the edges of the DAG.

An interlock is equivalent to use of a global variable. So if we implemented a similar optimised sort we would pick instruments that a lot of others depended on and perhaps were the head of a chain of a number of instruments each dependent on the last.

We know that if an instrument has no dependencies then we can play it now, if we have an available processor. We see that the goal of providing the largest number of instruments to pick at the next step means having a greater amount of parallelism at that step.

So at each `k` cycle when we receive our list of instruments to play we can build a DAG and perform the sort. When we have multiple instruments available at a step of the sort, we can perform them in parallel. This is in many ways similar to the computation in Jade's withonly do construct, the creation / running of live tuples in Linda and the performance any operation with its operands available in data-flow (as discussed in Section 2.5).

2.7 How do we Parallelise A Program?

Now that we have considered how some other semi-parallel languages work how will we implement parallelism? How do we make serial code, parallel? We have previously discussed adding parallel constructs to the Csound interpreter to do this, but what do we add? There are several things to consider:

- How do we tell parts of the program they should be executing in parallel?
- How do we wait for the results of a concurrent computation?
- When different parts of the program are executing concurrently how do we preserve the serial semantics of those parts? i.e. how do we ensure that only one of those parts can access shared data at a time?

2.7.1 Message Passing

The first possible solution is Message Passing. In this paradigm a program consists of multiple processes who communicate by sending messages to each other. This is generally used for MIMD machines with distributed memory, in which sending messages is the only way to communicate. It can also be used with shared memory machines. We are going to look at the Message Passing Interface (MPI) specification (Mes, 2008).

To use MPI in our program we need to call the MPI_Init function after which multiple processes are started. We can then determine the nature of our environment with with calls to MPI_$*$ functions. Some information we might need is how many other versions of us are there and what number are we. We can then use this to decide what to do, for example whether we are the producer or the consumer. We then use calls to variations of MPI_Send and MPI_Recv to communicate with the other versions of ourselves. More complex functionality that is commonly required like MPI_Reduce and MPI_Barrier (to synchronise processes) also exist. When we are done we call MPI_Finalize to wait for the other processes and exit.

Waiting for results of concurrent computations is handled in the receiving of a block of data from another process. The current computation will block until that one is finished at which point we can access the data that has been passed to us.

We can ensure the serial semantics by specifying who is communicating with whom ensuring an ordering on accesses to data in each process.

2.7.2 Threads and Locking Synchronisation

Pthreads[4] in the standard C Library provides parallelism inside a process. Each thread has its own stack, but they share the heap with every other thread.

Thread Create/Join

We create threads by calling a function, passing in an initial function that we want to run in parallel. After the other thread has been created this function returns and we continue in the current thread. If at some point we want to wait for the other thread to finish before we go on with our computation we can join the other thread. This will wait for the thread if it is not done or continue straight away if it is. The relevant functions in Pthreads are pthread_create and pthread_wait.

In Listing 2.4 instruments 101, 102 and 103 are playing at the same time. Rather than calculate the output for each instrument for each sample sequentially we could start some of the instruments in threads (two for 101, and one each for the others) calculate our instruments then wait for the threads to complete. We can add the results and write this

[4]For further information run `man pthread`

Listing 2.4: A Snippet of a Score Definition

```
1 ;inst        start    duration
2 i 101        0        1
3 i 101        0        1
4 i 102        0        1
5 i 103        0        1
6 i 104        1        1
```

to the output for that sample.

We should also note at this point that creating threads is not free. As a rough estimate for our test computer the time between the call to create a thread and for it to start executing is $90\mu s$ (Apple Inc., 2008*a*). If we create a thread to start executing a function which could be evaluated in serial in less than $90\mu s$ and we will need the results within 90 + (function computation time)μs then we have increased the time it takes our program to run.

Locking

We may want to set a variable to a value and then later on read back that variable. How do we ensure that in between the write and the read no one else has changed the variable?

The Pthreads API provides support for Mutexes. When executing a function before writing to the variable we would lock the mutex. When we have finished with that variable we would unlock it. If another version of ourselves executing this function tries to lock the mutex while we have it locked they will be forced to wait until we unlock the mutex. We only referred to mutexes in the context of executing the function, nothing stops us from just accessing the variable. It is our responsibility to ensure all access to goes through the mutex first.

To ground this in an example we are varying the amplitude of our instrument based on the number of times it has been used (Listing 2.5). We do not want two instances of this instrument to both alter the global variable gctr and then both use the resultant value. The amplitude would drop by double the expected amount instead of a steady decline. instr 101 has side effects and by definition side effects of two instances of 101 will not be mutually exclusive. This is a very simple example which is not parallelisable, however it may be executing along with several other instruments whose output could be calculated concurrently.

To fix this we would wrap lines 7 & 8 (in Listing 2.5) with a call to lock a mutex for gctr and then to unlock. We will ignore the handling of out for the moment.

If we know that two blocks of code have side effects that are mutually exclusive then we can just execute the blocks serially rather than locking variables. If we are expecting one block to use a variable before another this will preserve our serial semantics.

Listing 2.5: A Snippet of an Orchestra Definition

```
1          instr    101
2
3          kpitch   = p5
4 ;        kcps     = cpspch(p5)
5          kamp     = ampdb(p4)
6
7          gctr     = gctr + 1
8          kamp     = kamp / gctr
9
10         kenv     adsr 0.05, 0.05, 0.8, 0.05
11         kamp     = kamp * kenv
12
13 a1      oscil    kamp, kpitch, 1
14         out       a1
15         endin
```

Barrier

A barrier is a synchronisation mechanism that can be built with locks (but is also included in Pthreads). A barrier is specified with a number of threads which it will wait for. It keeps a count of how many threads have arrived. When a thread arrives it will block if less than the number expected have arrived at the barrier. If the thread arriving makes the count equal to the number the barrier is waiting for, the barrier will unblock all the threads waiting at it. Barriers can be used in a similar way to a thread join except at the end all the threads are still running.

In our example Listing 2.4 (Page 14) we could start two threads partition the instruments across them and place a barrier at the end. Once all the instruments have played both threads will reach the barrier and could then proceed to start playing the next cycle.

2.7.3 Conclusion

Both implementation styles provide the required features to implement parallelism in Csound. However Pthreads is the better option for several reasons.

Single Process Csound is a written as a single process and so adding parallelism inside that process is a good fit.

Current Usage Csound already features some usage of Pthreads. So the risk of introducing a new library to link to is avoided.

Data Structures Csounds data structures are all allocated dynamically. Passing these between processes in MPI would be expensive and ultimately just attempting to mimic the Pthread style of implementation.

Pthreads is a better fit for the Csound project, and the work to rewrite Csound would be too high risk for this project.

The main benefit of MPI would be the ability to run on distributed memory MIMD machines. However as stated in the introduction (Section 1.2 on Page 1) we are more focused on the possible speedup for consumer machines. Finally the interesting parts of the project—determining mutual exclusivity—should work across multiple implementations of parallelism. So if distributed memory becomes a priority in the future some work will already have been done towards it.

2.8 Bath Concurrent Lisp Machine

Finally we reach the most similar project to our own. The Bath Concurrent Lisp machine is a MIMD machine with distributed memory[5] built in the 1980s (Fitch, 1983). The compiler is automatically parallelising, taking serial lisp code and annotating it with parallel constructs. The methods used to do this are detailed in the following subsections.

Further details of the implementation are specified in the papers referenced but are not summarised here. These refer to implementation of concurrent Lisp and similar details which are not relevant to this project.

2.8.1 Data Flow Analysis

The Data Flow Analyser (DFA) built a closure for each entry point into a module.

From a single closure we can find how a function uses non-local variables. They are split into three groups (Fitch, 1983):

read read only
write written before being read
read-write read before being changed

A quadruple contained these sets for each function. The final unspecified component in the tuple stored a flag for whether the function was 'hard'—its side effects could not be determined.

As the flow analysis is done some functions will not have been analysed when they are used. Algebraic forms are used here which are back substituted into at the end of the analysis (Fitch, 1988). So functions that call other functions inherit their side effects. So if a function calls a 'hard' function it too is marked 'hard'.

There are added complications relating to analysing Lisp code such as access to property lists and other non-simple data. These are marked as 'hard' because the DFA is unsure

[5] it did however feature windows of memory writable by other processors allowing communication.

what the effects were. The inheritance of this characteristic made the problem worse. The solution to this is improving the DFA so it could more accurately determine the side effects and hence not mark the function as hard. For example building a list backwards and using the destructive list reverse operator nreverse appeared to be 'hard'. However a replacement of nreverse with another function without side effects that was not 'hard' solved this problem. Due to the inheritance any function marked 'hard' because of just a call to nreverse had the flag unset. There are other examples of further issues such as vector access (Fitch, 1989).

With the side effects of functions known, the code was annotated with the parallel constructs in Lisp (fork and join).

2.8.2 Complexity Analysis

Allowing all the parts of the program that are safe to run in parallel to actually run in parallel is naive as we saw earlier (Section 2.7.2). So this component assigns a weight to each function which is used to determine whether it is worth running in parallel (Marti, 1987). A cut-off value is defined whereby functions above this were significant and worth evaluating in parallel. Those non-significant functions are evaluated in serial as normal.

A by-product of a complete data flow analysis of the program is a complete call graph. This along with a static analysis of the program is used to determine the weights. Each of the primitive Lisp functions has a weight assigned. When calculating the weight for a function in the program:

composite the weight of the called function is added to the total
iteration the weight of the body of the loop is multiplied by `8`.
recursion the total weight of the function is multiplied by eight.

The use of the number eight is based on MU5 measurements. All weights are relative to the primitive function cons.

The results of this analysis were experimentally verified with a positive outcome aside from iteration. It was found that while the value `8` worked on average occasionally a program would be compiled for which this was an actively bad value. A feedback system was considered as a solution to this, but the program would always have to run once with the bad value—the first time—before this would take effect.

2.8.3 Horizontal Concurrency

The type of concurrency exploited in the Bath Concurrent Lisp Machine is referred to as horizontal (Marti, 1981). This type of concurrency appears in function calls in languages with applicative order function evaluation like the version of Lisp targeted. When two arguments to a function are mutually exclusive then we can evaluate them concurrently before passing the result to the called function.

We say:

$a << b$ when evaluating a can affect the value of b so a must be evaluated before b.
$a == b$ when a and b can evaluated in any order because they are mutually exclusive.

If in a function call (F $a_1 \ldots\ a_n$) there exists some groups i–k where $a_i == a_{i+1}, \ldots, a_{k-1} == a_k$ and $i \neq k$ then there is exploitable concurrency in those arguments. If the weights of the functions in i–k are significant they were evaluated in parallel (Marti, 1980).

2.8.4 Conclusion

We can see several useful ideas in the implementation. Specifically the idea of the Data Flow Analysis to determine mutual exclusion and the complexity analysis to determine if the program part is worth executing in parallel.

The relative simplicity of Csound compared to Lisp should ensure we can avoid some of the pitfalls identified such as:

- There are no loops in Csound so we do not have the unknown total iterations problem.
- There is no non-simple data that can force a DFA to regard a program part as 'hard'.
- There is no composing of other program parts (calling functions) so we can avoid the inheritance of variable usage.

The use of the DFA and complexity analysis at compile time should ensure we have a reasonable runtime for the first run, which would not be obtainable with dynamic analysis. With the simplification compared to Lisp we should be able to reasonably give assurance that the program will compute no slower in parallel.

We still expect to have to tune a DFA depending on the results it produces (see nreverse problem, Section 2.8.1).

2.9 Conclusion

We have identified a means of implementing parallelism and seen some previous automatically parallelising compilers. We intend to implement a Data Flow Analyser and a Complexity Analyser for Csound. We will use this to determine which program parts can be run at the same time and whether it is worth it to do so.

The DFA will use the semantics of the assignment operator in Csound orchestra language and the order the instruments are called in the score to determine the dependencies of instruments. The complexity analysis will use information about the opcodes in an instrument to determine whether the output of instrument is significant. We will build a DAG

from the dependencies and start to sort it. At each step of the sort it we are presented with several instruments it means they are currently mutually exclusive. If they are also significant we will use the Pthreads API to execute them in parallel. Once we finish performing all the instruments in the `k` cycle we will synchronise the threads before moving onto the next `k` cycle.

Chapter 3

Requirements

In this chapter we discuss the requirements to be met for the outcome of the project to be considered a success. We collected the requirements in an iterative process, developing the initial set from the Csound maintainer (our supervisor). From there, as we worked on the project we introduced requirements that specified extra steps we could take. For instance in Non-Functional Requirement 1.1 we knew we had to maintain a serial semantics access order to global variables from the work seen in the literature review. We then saw that we could make a more specific requirement for a certain condition in Non-Functional Requirement 1.1.1. This condition allows us to rearrange global variable access so long as the output is maintained.

For these requirements "Must" refers to those that are mandatory and "Should" to those which can be omitted if time does not allow for them. The success of the project is determined by the number of must requirements met, with should as an added bonus.

3.1 Functional Requirements

(1) Implement a system to determine the dependency of an instrument on global variables.

  (1.1) This must support numbered and should support "named" instruments.

  (1.2) This must recognise all uses of global variables that cause dependencies. These are: the global is used on either side of an opcode, inside a conditional statement, or in an assignment.

  (1.3) This must support a reasonable range of orchestras. Reasonable as limited by the time we have to implement and the prototype status of this project.

(2) Use the system in Requirement 1 to build a system which determines the dependencies of an instrument upon other instruments.

(3) A mechanism to perform instruments in parallel must be implemented.

  (3.1) This must use pthreads.

  (3.2) This should be portable to all the platforms Csound currently supports.

(4) A facility for specifying the weights of opcodes should be provided.

(5) A facility to calculate the weights of opcodes should be provided.

3.2 Non-Functional Requirements

(1) The semantics of the input program must be preserved.

  (1.1) Access to global variables must occur in an order which is semantically equivalent to that specified by the programmer. This allows for the following situations:

    (1.1.1) Rearrangements of increments into the variable. So long as the value is preserved between reading it to calculate the increment and writing it back in.

    (1.1.2) Blind writes where we do not care what is currently in the variable.

    (1.1.3) Multiple reads occurring simultaneously.

  (1.2) Access to shared resources (such as output) must be protected.

  (1.3) Rendering a piece of music on more than one processor must produce the same output as rendering it on one processor. This will be verified sample by sample.

(2) The time taken on single processor machine to render a piece of music must not be affected within reason. This allows for the extra time to analyse the input program.

(3) Performance on a machine with more than one processor should be greater than on single processor machine. Allowing for input programs which feature no available parallelism.

  (3.1) Dispatching of tasks should be optimised to take advantage of the available number of processors and the varying amounts of communication.

  (3.2) Tasks should only done in parallel if a performance gain would result.

(4) The process of parallelising the program must be invisible to the programmer.

  (4.1) No additions can be made to the Csound languages.

  (4.2) The programmer must not be required to give any ‘hints’ about where to parallelise.

Chapter 4

Test Plan

We have assembled several pieces to test the full range of computation and parallelism that the parallel version of Csound will meet in practise.

priest Electric Priest by Tobias Enhus. The piece with the heaviest amount of computation and no available parallelism. It features a reverb instrument at the end of the `k`-cycle with the associated global variable that all instruments increment.

t07 A very simple piece which is very light on computation but features large amounts of parallelism. It makes no use of global variables.

t09 An alteration to t09 to introduce a reverb instrument. Each other instrument increments into the global reverb variable. So this piece features no available parallelism.

xanadu X A N A D U (short version) by Joseph T. Kung. Moderate to heavy amounts of computation with full parallelism available. No global variables are used.

trapped Trapped in Convert by Richard Boulanger. Moderate to heavy amounts of computation with some parallelism available. Traditionally used as a benchmark by the Csound community. This piece features two "reverb" instruments: a reverb and a delay. Some instruments increment into one, others into both and some into none.

Piece	Time (s)
priest	54.11
t07	7.83
t09	10.79
xanadu	44.40
trapped	17.68

Table 4.1: Test Pieces Serial Performance Times

The execution times for serial Csound for each of our test pieces are listed in Table 4.1. To be successful we need to achieve times which are less than these for each of the test pieces.

We also need to test the accuracy of the parallel program. We will validate this using a program to compare the results of execution in serial and parallel, sample for sample. We will use this to determine whether our parallelism implementation is correct and if not, where the problems lie in it.

Csound contains a timer for the computation of the orchestra. Ad-hoc timing will be conducted using this. More structured timing where multiple samples are taken is done with a script which performs all the tests, with repeats as specified by the sample count. The times in this script are taken from before the call to create a new process, to after that process has returned its return code. These two timing systems are not comparable, however we will only compare either like with like, or the difference in times will make difference in overhead irrelevant.

All of our testing is done on a dual core Mac OS 10.5 machine. We compile Csound with gcc 4.0[1], which does not have the `__sync_lock_test_and_set` atomic instruction used to build the spinlock in Csound. We use the Mac OS specific `OSAtomic` spinlocks instead. We compile with the debugging symbols and default optimisation.

In addition to the timing we also when required, use the Shark performance analysis tool[2] provided with the Mac OS X Computer Hardware Understanding Development (CHUD) Tools (Apple Inc., 2008*b*). We use two of its facilities: time profiling and system tracing.

Time Profiling provides a statistical sample of the programs execution by recording a sample at specified intervals (we use a 1 ms sample interval). These samples contain the process ID, thread ID, program counter and call stack. These details are used to reconstruct what was executing when the sample was taken. We use this to see where in Csound we are spending most of our time. This allows us to focus our attention on making improvements which will actually make a difference.

System Tracing records an exact trace of system level events including thread scheduling decisions. We use this to determine what threads are running at what time, investigate why a thread was not scheduled and what they are blocking on.

[1] version i686-apple-darwin9-gcc-4.0.1
[2] version 4.6.1

Chapter 5

Design and Implementation

There are three main components to the implementation of parallelism in Csound. They are:

Semantic Analysis identification of parallelism constraints.

Dispatching which processor to do the work on, and in which order.

Weighting System loading, calculating weights of opcodes and instruments.

We discuss each of these and the major iterations within. We have tried several designs within these concepts and identifying the problems with the less effective designs is important for future work.

Following these sections we explore a new combined dispatch design, performance on a four processor machine and the effects of compiler optimisation.

Listing 5.1 is an example orchestra which we will use to discuss the various designs. We have left in just the global variable accesses. In the serial version of Csound it would be played in the order 1, 2, 3, 4, 98, 99. In all of our examples we will assume that one of each instrument is playing.

Before we begin we will first describe the initial implementation of parallelism in Csound.

5.1 Initial Design

The functions which are relevant to our implementation are:

`csoundPerform` is the main loop around the performance.

`sensevents` is called to determine what instruments should be playing at this point in time.

Listing 5.1: Example Csound Orchestra

```
1          instr 1
2          ...
3  garvb   = garvb + asig
4          endin
5
6          instr 2
7          ...
8  garvb   = garvb + asig
9          endin
10
11         instr 3
12         ...
13 garvb2  = garvb2 + asig
14         endin
15
16         instr 4
17         ...
18 garvb2  = garvb2 + asig
19         endin
20
21         instr 98
22 asig    reverb garvb
23 garvb   = 0
24         endin
25
26         instr 99
27 asig    reverb garvb2
28 garvb2  = 0
29         endin
```

`kperf` plays the instruments decided upon by `sensevents`.

`kperfThread` is a loop around playing the current group of instruments. It synchronises with `kperf` before and after playing the group. Otherwise it runs until told the performance is complete.

We consider each performance in the main loop in `csoundPerform` as a `kperformance`. At every `kperformance sensevents` is called, setting up the instruments necessary. `kperf` is then called which actually performs the instruments. When inside `kperf` a barrier is passed before performing instruments, and one when they are done. These synchronise with `kperfthread` which has barriers in similar positions.

`csound_orc.y` is the yacc / bison grammar file for the Csound orchestra language. As usual in a yacc / bison grammer specification file, actions can be added which are performed when a rule is applied.

Determining what work is done on what thread is performed by `partitionWork` which splits the work across threads based on performing multiple instances of the same instrument at the same time.

The number of threads to use is determined by the `--num-threads` argument to Csound. This specifies the number of extra threads to start. These extra threads are worker threads which perform the instruments. The main thread decides what instruments to play at a `kperformance` but does not do any work to perform them.

5.2 Semantic Analysis

Semantic analysis follows a similar concept to the data flow analyser described in Section 2.8.1 on Page 16. We need to identify the global variables each instrument uses. We then need to decide whether they are read or written to.

We considered two different implementations: a tree walk and using the parser. We decided on a hybrid version which would perform a sub-tree walk when called in certain places in the parser. We decided this was the best combination for ease of implementation and its self-documenting nature.

We created a list of instrument data structures. Each instrument structure contained a set of global variables that were read, and those written by the instrument. After each sub-tree walk the global variables found were added to the appropriate set.

Figure 5.1 on Page 27 illustrates what our example orchestra (Listing 5.1 on Page 25) looks like in this structure.

We later added the read-write category of global variables as described in Section 5.3.4 on Page 36.

5.3 Dispatching

We discussed (in Section 2.6 on Page 11) an approach—based on instruction scheduling—to deciding what work to do in parallel based on the dependencies we have just discovered in Section 5.2.

We initially just implement a simple topological sort as a proof of concept. We know from semantic analysis which instruments use which global variables, we also know that instruments are played in order of increasing number. With this information we can build a dependency graph of the currently playing instruments. We make an edge going from instrument b to another a if $a < b$ and:

- a writes and b reads the same global variable.
- a reads and b writes the same global variable.

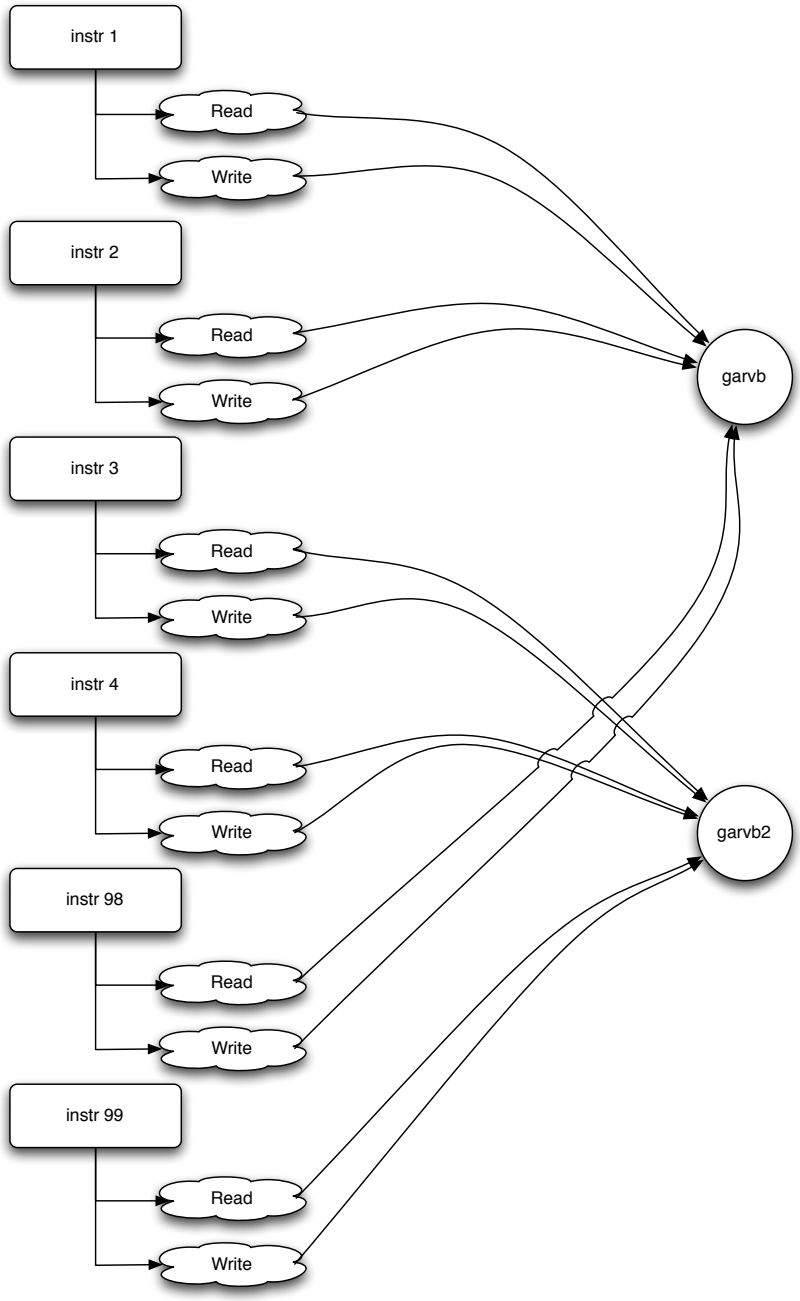


Figure 5.1: Semantic Analysis Data Structure of Sample Orchestra

- a writes and b writes the same global variable. We must ensure any instrument $c > b$ sees the right value in the global variable a and b share.

We then have a DAG which we can perform a topological sort on to get a semantics preserving order in which we can play the instruments. We can also go further and perform a customised topological sort based on the instruction scheduling ideas in Section 2.6 (Page 11) in an attempt to produce an optimised order to perform the DAG in.

There are 4 major iterations in the design of the dispatch component of Csound parallelism. They are:

(1) Dynamic Non-Caching
(2) Component Non-Caching
(3) Dynamic Caching
(4) Component Caching

The dynamic designs follow the concept of creating and sorting the DAG closely. The component designs are an extension which uses these ideas but finds a list of instruments for each processor to play. We will discuss each of them: the reasons for that design and whether it was successful.

5.3.1 Dynamic Non-Caching

The initial implementation of the dynamic design described in Section 5.3 used an object like model to represent the DAG with a struct for the DAG handle containing pointers to a set of DAG nodes. Each of these DAG nodes was itself a struct containing a set of pointers to other DAG nodes which represented its edges.

Each thread calls `consume` on the DAG while there were remaining instruments to play. `consume` searches for roots (a node without any incoming edges) that it thread can give to the thread. After the call to consume the thread played the instrument and then called `consume_update` which removed the node from the graph and any edges starting from it. It then updated the available roots. This was a destructive operation, de-allocating memory.

It was necessary to split out the `consume` and `consume_update` as the other threads should not be able to start playing an instrument depending on the consumed instrument until it has finished being consumed.

The `consume` and `consume_update` functions preserved serial semantics by using mutexes to protect access to the DAG data structure they operate on. There was one mutex for the DAG handle.

We need a method to keep hold of the handle to our DAG over `kperformances` which use the same instruments. Otherwise we will have to repeatedly build and destroy the DAG after each one. We could implement a DAG modification algorithm which can modify a

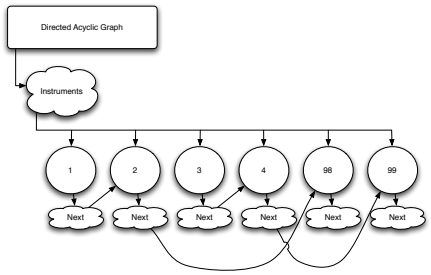


Figure 5.2: Dynamic Design Data Structure For Example Orchestra

Piece	Time (s)	
	Serial	Parallel
priest	54.11	250.55
t07	7.83	238.96
t09	10.79	924.26
xanadu	44.40	1056.73
trapped	17.68	141.88

Table 5.1: Dynamic With Mutex Non-Caching Performance Times

DAG to include the new instruments with a logarithmic upper bound. However this is still a lot of allocation and deallocation when the DAG does not change at all for large numbers of `kperformances`. Our current dynamic design is also not conducive to caching due to the the destructive method by which `consume` and `consume_update` operate. It was designed as a means to test the overall idea. We intended to implement caching later but at this point the DAG for a `kperformance` was created at the start of the performance and then destroyed at the end.

Figure 5.2 illustrates our example orchestra (Listing 5.1 on Page 25) in the form of the data structure described previously.

Table 5.1 shows that this implementation is considerably slower than the Csound serial version. We assumed that the building and destroying of the DAG at each `kperformance` was probably the largest overhead. However we needed to get some evidence to show this.

Table 5.2 on Page 30 contains the time profile of this implementation. We have selected just the symbols in which more than 1% of samples were found. We highlighted those that are related to the construction and destruction of the DAG. They account for 57.5% of the total time.

Number of Threads

We initially assumed that if we have two processors we should use two threads. How accurate is this? On our test machine should we use one thread for each processor, or

Total (%)	Symbol
24.0	`mfree`
21.2	`csoundLockMutex`
17.0	`mmalloc`
5.1	`csoundUnlockMutex`
4.3	`set_alloc`
3.5	`csoundWaitBarrier`
2.1	`out`
1.8	`dag_build_initial`
1.8	`dag_build_edges`
1.8	`kperfThread`
1.8	`set_dealloc`
1.7	`set_update_cache`
1.5	`set_element_get`
1.4	`set_is_set`
1.2	`set_count`
1.2	`set_intersection`
1.2	`osckki`

Table 5.2: Dynamic With Mutex Non-Caching Time Profile

more. We can not test using less as that would be running in serial. We experimented with 2–4 threads and the results are contained in Table 5.3.

Piece	Time (s)		
	2 Threads	3 Threads	4 Threads
priest	250.55	252.56	261.79
t07	238.96	268.10	286.05
t09	924.26	985.72	1023.98
xanadu	1056.73	1167.80	1211.37
trapped	141.88	183.96	186.32

Table 5.3: Performance Times for Varying Number of Threads for the Dynamic with Mutex, Non-Caching Design

We can see from the results that using as many threads as processors is the fastest approach. Before we attempted any other optimisations we first experimented with our other design.

5.3.2 Component Non-Caching

Component non-caching uses the concept of identifying the connected components in a graph and partitioning non-connected sub-graphs across threads. This means we can calculate the serial order to play instruments in on each thread before the `kperformance`

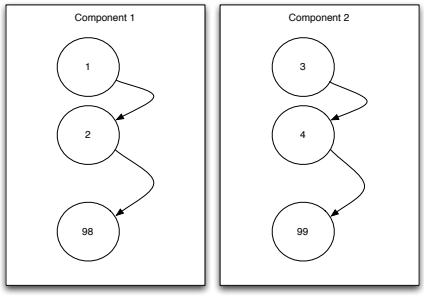


Figure 5.3: Connected Components of the DAG of our Sample Orchestra

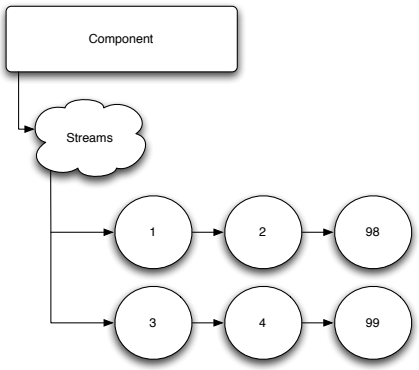


Figure 5.4: Component Streams for Sample Orchestra

begins. The advantage of this is we do not need to do any locking inside a `kperformance` as we will not need to communicate with anyone else.

The partitioning into connected sub-graphs was achieved by creating sets of nodes. Starting with a set for each node. If there was an edge between elements of two sets, the sets would be replaced with a new one formed from the union of them. This continued until there were no changes for one iteration through all the sets. If the number of sets remaining was greater than the number of threads, the first two sets would be replaced by the union of them repeatedly until the number of sets was equal to the number of threads. Figure 5.3 illustrates these connected sub-graphs for our example orchestra (Listing 5.1 on Page 25).

Once we had a group of connected sub-graphs we iterated through them performing a topological sort on each. We collected these orders of instruments—we call a stream—into a data structure. We now had streams of instruments where the number of streams was less than or equal to the number of threads. Figure 5.4 shows the streams created from the connected sub-graphs in Figure 5.3.

At each `kperformance` each thread selected a stream according to the thread index. If the number of streams was less than the number of threads, then threads which did not have a

Total (%)	Symbol
34.9	`mfree`
28.3	`mmalloc`
5.3	`set_update_cache`
5.2	`set_alloc`
2.7	`csoundWaitBarrier`
2.5	`set_dealloc`
1.9	`set_intersection`
1.7	`set_element_get`
1.4	`dag_build_initial`
1.4	`edge_nodes_build`
1.3	`set_is_set`
1.3	`set_add`
1.1	`set_count`
1.0	`set_element_ptr_eq`

Table 5.5: Component With Mutex Non-Caching Time Profile

stream would proceed to the synchronisation barrier at the end of the `kperformance` and wait for the others.

The steams were implemented using the set data structure taking advantage of its insertion order preserving semantics. When an instrument was removed memory de-allocation occurred in the set. So again we could not implement a cache around this design.

We can see from the execution times in Table 5.4 that the component design is slower than dynamic (Table 5.1 on Page 29) and serial (Table 4.1 on Page 22).

Piece	Time (s)		
	Serial	Dynamic	Component
priest	54.11	250.55	594.10
t07	7.83	238.96	241.82
t09	10.79	924.26	765.42
xanadu	44.40	1056.73	1182.84
trapped	17.68	141.88	350.97

Table 5.4: Component With Mutex Non-Caching Performance Times

Table 5.5 on Page 32 again contains the time profile of this implementation. The samples are again taken at 1 ms intervals and we have selected symbols with greater than 1% of the samples. We have highlighted the symbols related to construction of the streams. This time the construction and destruction of the streams takes up 87.3% of the samples.

5.3.3 Dynamic and Component Comparison (Non-Caching)

Before our comparison we have a look at some general improvements that can be made to both designs.

Spinlock Barriers

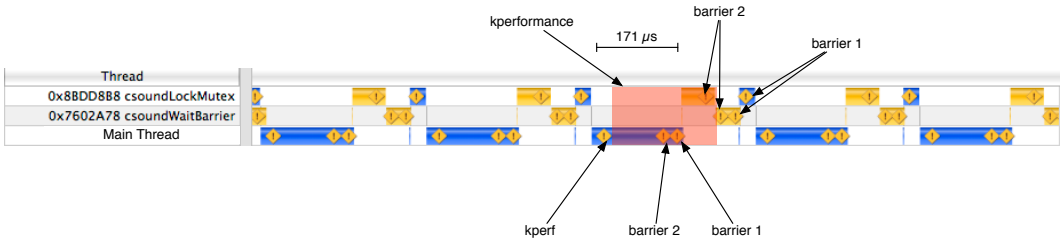


Figure 5.5: Dynamic With Mutex Non-Caching System Trace

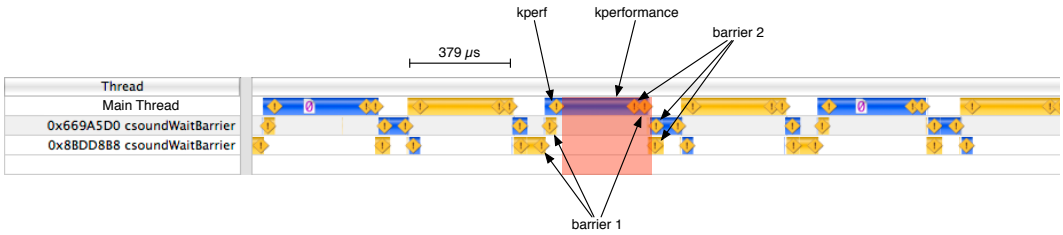


Figure 5.6: Component With Mutex Non-Caching System Trace

We inspected the system trace of both the dynamic and component implementations. A screenshot for each is supplied in Figure 5.9 and 5.6 respectively. We saw the threads are essentially running sequentially, we suspect that we are spending some time waiting for our threads to be scheduled. We implemented a barrier using spinlocks and used this to synchronise at the start and end of the `kperformance`.

This barrier used a lock for the barrier data and an array of locks, one for each thread. When a thread enters it checks if enough threads have entered and if they have unlocks all the other threads. If they have not it blocks on one of the array of locks. There is an issue if one of the threads once unblocked enters the barrier again before another of the threads can leave. However as the barriers are used in pairs the early thread would be blocked at the other barrier.

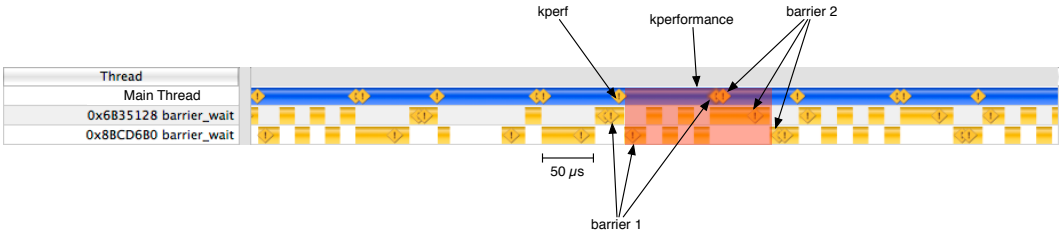


Figure 5.7: Dynamic Non-Caching System Trace

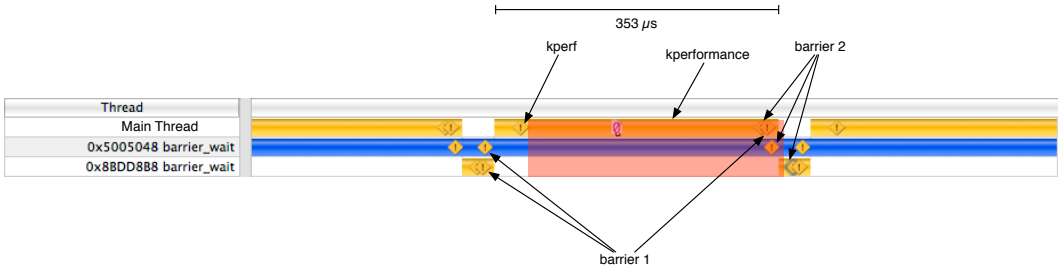


Figure 5.8: Component Non-Caching System Trace

We looked at the system traces for the dynamic and component implementations running with spinlock barriers (Figure 5.7 and 5.8 respectively). We can see that at the main thread is running all the time or at least most of it. However there is still some sequential thread scheduling. The more continuous running of main thread ensures that after a `kperformance` we can begin preparing for the next one as soon as possible, so we keep this change.

Piece	Mutex Time (s)		Spinlock Time (s)	
	Dynamic	Component	Dynamic	Component
priest	250.55	594.10	227.11	558.32
t07	238.96	241.82	194.95	226.31
t09	924.26	765.42	908.40	731.92
xanadu	1056.73	1182.84	819.78	1125.95
trapped	141.88	350.97	111.74	310.28

Table 5.6: Dynamic vs. Component With Spinlock and Mutex Barriers Performance Times

The times for the dynamic and component designs with spinlock barriers are in Table 5.6. The dynamic version with spinlock barrier is faster in all tests than the mutex barrier version. `Xanadu` in particular sees a 200 second reduction in computation time. The component version is only slightly faster with the spinlock barrier than the previous mutex barrier implementation.

Between the spinlock barrier versions dynamic is faster in all but the light computation, zero parallelism test (`t09`). Even this implementation is still dramatically slower than the serial version of Csound.

Number of Threads Definition

We saw previously that worker threads execute sequentially. To try and get then to run in parallel we changed the definition of `--num-threads`. From specifying the number of worker threads to specifying the total number of threads. We then moved some work into the main thread: once it passes the first barrier and before the second it will perform the same work as a worker thread. This means that when we get one thread for each processor and each will do work. The threads should each be running for as much time as possible. This avoids relying on the scheduler having to select the right threads to run before and after the barriers are passed.

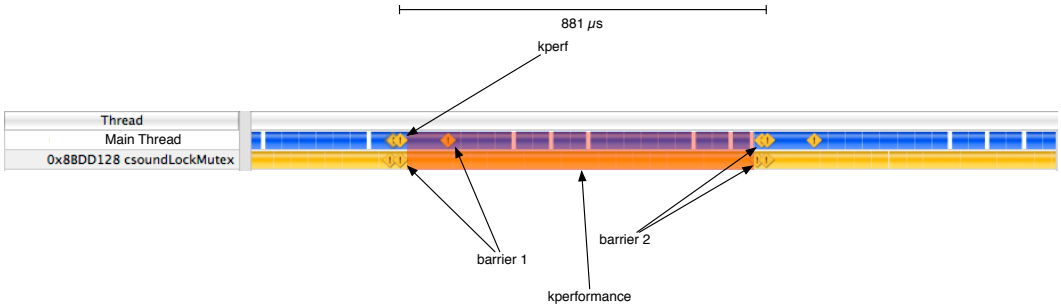


Figure 5.9: Dynamic with "Working" main thread Non-Caching System Trace

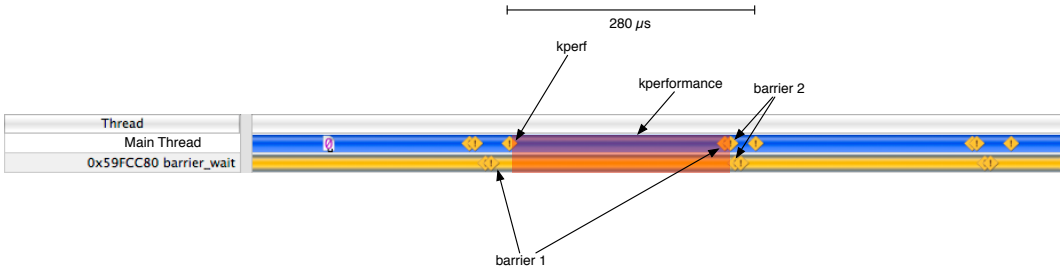


Figure 5.10: Component with "Working" main thread Non-Caching System Trace

The system traces for the new definition of `--num-threads` are in Figures 5.9 and 5.10 for the dynamic and component respectively. We can see that both threads run virtually continuously.

Piece	Time (s)	
	2 Workers	Working Main Thread and 1 worker
priest	227.11	211.64
t07	194.95	231.63
t09	908.40	892.97
xanadu	819.78	987.38
trapped	111.74	100.35

Table 5.7: Dynamic Non-Caching vs. Dynamic "Working" main thread Non-Caching Performance Times

Piece	Time (s)	
	2 Workers	Working Main Thread and 1 worker
priest	558.32	524.22
t07	226.31	209.96
t09	731.92	734.36
xanadu	1125.95	1080.00
trapped	310.28	272.42

Table 5.8: Component Non-Caching vs. Component "Working" main thread Non-Caching Performance Times

The change to the definition of `--num-threads` yielded some further improvements for dynamic with 3 of 5 tests improved (Table 5.7). This is not a clear victory and we will revisit this later when we look at the cached dynamic implementation in Section 5.3.5. Component was slightly more clear with reduced times in all but one test (Table 5.8).

Conclusion

With both of these changes we see that Csound now greedily takes all available processors for as much time as possible. We justify this on the basis that we are interested in the maximum speedup possible.

As we saw in Tables 5.2 and 5.5 (on Pages 30 and 32) the times taken to build the DAGs/streams at each `kperformance` still dwarfs the computation involved for the actual instruments. Before we attempt any further optimisations we need to build versions of the dynamic and component designs which are amenable to caching.

5.3.4 Read-Write Global Variables

We consider an issue raised by our supervisor regarding orchestras which use a global reverb at the end of the `kperformance`. Each instrument reads the global and then writes into it,

forcing the performance to occur in serial. As we are only incrementing into the global it does not matter which order we perform the instruments in. So long as only one instrument is performing the incrementation at any one time. Otherwise two instruments could read the global variable, calculate the new value by adding its output into an intermediate. When they both write back the of the output from the first instrument to write will be lost.

Figure 5.11 on Page 38 illustrates the how the semantic analysis structure of our sample orchestra (Listing 5.1 on Page 25) changes with the implementation of read-write global variables.

In the dynamic design the protection during incrementation is relatively simple. We insert spinlocks around the incrementation preventing other instruments from accessing the same global at the same time. When finding dependencies we also keeps a list of read-writes to global variables. Instruments which read-write the same variable are allowed to occur at the same time. Similar rules to the original dependencies exist such as if another instrument reads or writes the read-write global then it must not occur at the same time.

When inserting the spinlocks we take care to perform this step before expanding expressions in the AST. Doing this allows us to just add the locks around one sub-tree and let the expansion take care of how many real opcodes the global variable should be locked for. Otherwise we may accidentally lock around the read and write separately and hence make the locks useless.

The component design is somewhat more complicated. We proposed several ideas:

- A recursive definition of dynamic and component structures. We have an initial dynamic structure which gives back component structures containing streams to do work on. After the stream is done we try to consume again. This provides the ability to perform the read-write instruments in parallel whilst allowing a synchronisation before the next read or write.
- A simplified version taking into account the fact that most reading of global variables is done at the end of the `kperformance` in reverb instruments. Before that instruments either read-write or do not use global variables.

The first idea is more general and would allow for all orchestras.

5.3.5 Dynamic Caching

The implementation of the previous dynamic design was the least complex. As component uses dynamic to build and sort its DAGs we decided to implement the new version first.

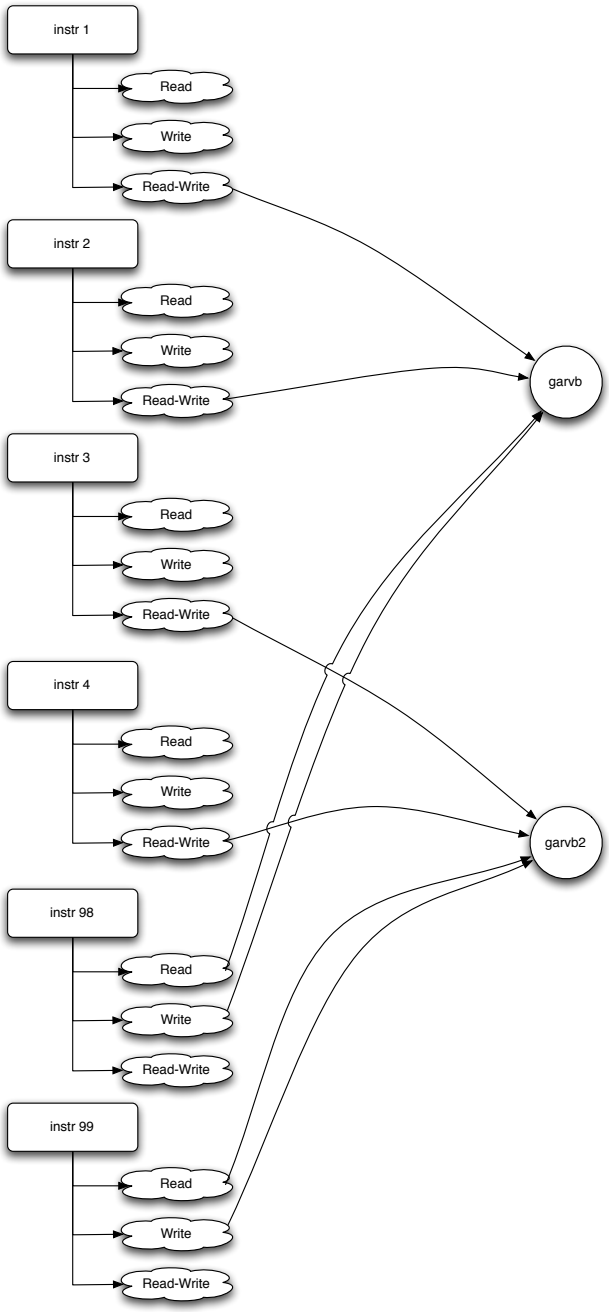
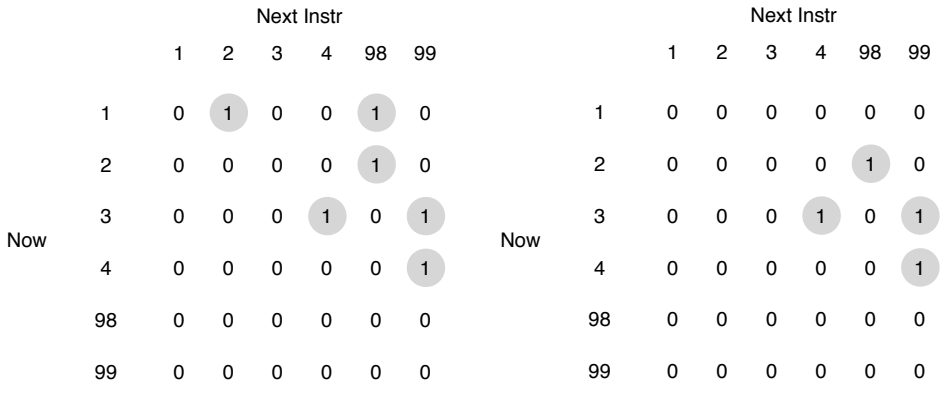


Figure 5.11: Semantic Analysis Data Structure with Read-Write Global Variables of our Sample Orchestra



(a) Matrix DAG Representation Starting State (b) After Playing Instrument 1

Figure 5.12: Matrix DAG Representation

Design

We represent the directed acyclic graph of instrument dependencies as a matrix, where each row and corresponding column represents an instrument. For an instrument's row 1s are placed in the columns of all the instruments which can follow that instrument. This leads to the orthogonal viewpoint that in an instrument's column a 1 indicates that the instrument in that row must be performed before the instrument the column represents.

The matrix was implemented as a 2D array stored in contiguous memory with an eye to the use of `memcpy` to reset the DAG. We kept an original of every mutable variable in the DAG, allowing us to easily reset them when the DAG needs to be reused.

Figure 5.12 illustrates this with our example Csound orchestra (Listing 5.1 on Page 25). Figure 5.12a shows the starting state of this matrix representation. A column with no zeroes means there is no instrument which must occur before the instrument represented by that column (i.e. it is a root). We search for a root instrument to play first. After we have played the instrument we remove the ones from the row representing that instrument. In Figure 5.12b after we have finished playing instrument 1, instrument 2 is now available to play.

The `consume` and `consume_update` functions from Section 5.3.1 (on Page 28) are updated for the new implementation. Again there is one lock to protect access to the DAG.

Blocking Consume

The first issue introduced by our new dynamic design was the use of polling on the availability of a root to consume. Table 5.9 (Page 40) shows a time profile of `t09` our simplest serial test case. We saw the hotspot of polling on consuming a root. When a root becomes

Total (%)	Symbol
57.7	`csp_dag_consume`
27.1	`csp_dag_consume_update`
8.9	`csp_dag_is_finished`
4.4	`kperfThread`
0.9	`csp_dag_is_finished`
0.7	`csp_dag_consume`
1.6	`kperf`
0.1	`barrier_wait`

Table 5.9: Dynamic Non-Block on Consume Time Profile

Total (%)	Symbol
43.1	`csp_dag_consume_update`
17.8	`csp_dag_consume`
7.0	`barrier_wait`
4.8	`osckki`
4.4	`out`
3.5	`nodePerf`
3.0	`addaa`
2.0	`buzz`
1.7	`aassign`
1.5	`foscili`
1.1	`reson`
1.1	`reverb`
1.0	`balance`

Table 5.10: Dynamic Block on Consume Time Profile

available the poller may be as far as a function exit, conditional branch and function call from getting to the root. We decided to change the consume function to be blocking so a thread entering it would only leave with a root or when the DAG was completely consumed. Table 5.10 (Page 40) shows the effect of this on the time profile.

Table 5.11 shows the time for the new implementation with and without the polling on our test pieces. The result is a an improvement in mostly serial cases with fully parallel cases being no worse. Obviously the blocking is an improvement we will keep.

Piece	Time (s)	
	Polling	Blocking
priest	725.25	176.83
t07	100.24	97.09
t09	1074.68	298.84
xanadu	377.68	377.15
trapped	144.98	85.50

Table 5.11: Dynamic Polling vs. Blocking on Consume Performance Times

Number of Threads Definition Revisited

Piece	Time (s)	
	2 Workers	Working Main Thread and 1 worker
priest	176.83	152.66
t07	97.09	81.23
t09	298.84	282.75
xanadu	377.15	325.60
trapped	85.50	62.02

Table 5.12: Dynamic Caching vs. Dynamic "Working" Main Thread Caching Performance Times

We left the issue of the definition of `--num-threads` open for further investigation (Section 5.3.3 on Page 35). The results in Table 5.12 demonstrate an obvious benefit to the new implementation. From this point on we use the new definition: the main thread does work and the number of worker threads is equal to the argument of `--num-threads` -1.

Cache

Now that we have a DAG representation that can be easily reset and we have solved some initial design issues, we need to construct our cache. The initial design of the cache was a simple list of entries pointing at the appropriate DAG structure. We identified cache members by implementing an equality operation between the lists of instrument numbers making up a DAG. New cache entries are inserted at the head of the cache list.

We kept a counter which decided on each cache fetch whether to first clean the cache before getting the appropriate DAG. When a DAG was returned its use counter was incremented. At each cache clean the age counter of each cache entry would be shifted right. If the age was beneath a certain point and the uses beneath a certain level then the entry would be removed from the list. These levels were picked so that new entries would survive eight cache cleans based on age before the usage came into effect. There is no empirical base for the numbers used.

Piece	Time (s)		
	Linear	Hash	Hybrid
priest	63.00	62.79	63.52
t07	8.93	8.40	8.09
t09	26.46	25.98	25.22
xanadu	37.67	38.62	36.76
trapped	20.50	20.80	20.53

Table 5.13: Dynamic Linear vs. Hash Caching Performance Times

The times for the linear cache version are in Table 5.13. All of the times show a significant improvement and are now much closer to the serial version of Csound. `xanadu` the fully parallel, moderate to heavy computation piece is now faster than the serial Csound version. With the introduction of caching the parallelism gained for `xanadu` outweighs the overhead of managing that parallelism. `t07` the other parallel piece is still on the other side of that equation.

We thought we could easily improve on the linear access time to the cache with a hash table based cache. Once this was implemented we tested again and the times are in the same Table (5.13) as the linear for comparison. We found some improvements but they are not clear cut. We considered that the instruments playing probably do not change for one second or so on average. Hence we will often be accessing the same cache entry—the most recent. This works well with the linear cache where this is first, but the hash version has the extra overhead of calculating the hash. We tried a hybrid method which keeps an extra last used cache as well as a hash cache (Table 5.13 again). This offers more widespread improvements, but `priest` is slower than either linear or hash. This is possibly caused by the more complex implementation of the hybrid with more conditionals, which may cause branch prediction issues.

We decided at to use the hash cache for the remaining tests as it gives us the best upper bound on fetching a DAG, even though it performs marginally worse.

Root Countdown

Piece	Time (s)		
	No Root Countdown	Root Countdown	
	1 lock no countdown	1 lock	2 locks
priest	62.79	61.72	61.88
t07	8.40	8.33	8.55
t09	25.98	19.32	18.30
xanadu	38.62	41.68	37.76
trapped	20.80	20.34	20.43

Table 5.14: Dynamic Countdown For Each Root Performance Times

As discussed in Section 5.3.5 the structure of our DAG means we can check each column for what instruments must be performed before we can play the instrument represented by the column. All the edges are in the upper right triangle and we never add edges. We do not need to actually remove the edges when updating or check an entire column is zero before identifying that instrument as a root. We can keep a count of how many instruments must be played for each instrument before we can play it. We need only then read the row after playing an instrument and if we find a one in a column decrement the counter for the instrument associated with that column.

We have reduced the identification of roots to a comparison of the counter with zero. We no longer change the matrix so we do not have to `memcpy` it from the original. This means less work to do when resetting a DAG before it is used when fetching it from the cache.

We are also now able to lock at a more granular level: one lock for the remaining instruments counters and one lock for the rest of the DAG.

Table 5.14 contains the times for both of the locking variants of root countdown. We can see a marked reduction for `t09` (the fully serial test). The DAGs associated with this piece have the property of each instrument played releasing just one more instrument to play. As we only lock the whole DAG when we find a root as opposed to for the whole of the `consume_update`, the other thread can get into `consume` faster. The other times are equivalent to the non-root counting version.

Counting Semaphore

We considered a possible issue with the implementation of block on consume next. It utilised one spinlock upon which threads would lock when they entered `consume`. When a root was found and set into the first root cache an unlock was performed releasing one thread. If another root was found it would have to wait for the thread just unlocked to reach the aforementioned part of `consume`.

If we wanted to unlock whenever we found a root inside `consume_update` we could do so, but this would require extra state to avoid a lock/unlock imbalance with `consume`. We want to be able to unlock whenever we find a root, so in a fully parallel case we can get as many threads into `consume` and getting roots soon as possible. Spinlocks only allow us to communicate that we want to let in one additional thread: if we unlock more than once the lock has the same state as unlocking once. What we wanted as a means to solve this is a way to count how many threads we should let in. We implemented a kind of counting semaphore to achieve this. The count in the semaphore represents the amount of parallelism currently available.

When a thread entered `consume` it would wait on the semaphore. When a thread found a root it would increase the size of the semaphore. At the end of `consume_update` a release would be performed. So if the first thread found three roots the semaphore count would be three, with the first thread taking one root and leaving two other threads to take the others. If there were no more instruments then the count would decrease back to zero as

each thread finished performing its instrument. Any additional threads entering consume after the first thread would be blocked until the last thread took its instrument. At that point we release all threads so they can determine the DAG is finished.

Piece	Time (s)	
	Spinlock on first root	Counting Semaphore
priest	58.50	64.53
t07	8.13	8.69
t09	17.07	19.29
xanadu	35.35	38.88
trapped	19.03	20.47

Table 5.15: Dynamic Spinlock updated on First Root vs. Counting Semaphore Performance Times

Table 5.15 shows the effect of the implementation of the counting semaphore was to reduce performance. The lock on the DAG serialises access to `consume`. So even if threads can get past the the `consume` semaphore faster, they cannot get an instrument and start playing it. We decided to keep the spinlock on first root for the remaining tests. This issue would probably be worth revisiting on a machine with more than two processors.

Read-Write Global Variable Implementation

We next implemented the read-write global variable optimisation discussed in Section 5.3.4 (Page 36) in the manner described therein for the dynamic design.

Piece	Time (s)		
	Spinlock on first root	Counting Semaphore	No Read-Write Globals
priest	33.68	33.88	58.50
t07	8.13	8.74	8.13
t09	11.10	12.00	17.07
xanadu	35.62	38.60	35.35
trapped	16.36	17.37	19.03

Table 5.16: Dynamic Read-Write Global Variable Spinlocks Performance Times

The introduction of the read-write global variables (Table 5.16 for a comparison) is the step which improves performance past serial Csound speeds for certain tests. `t09` previously featured no parallelism, but now is mostly parallel except for the last instrument bringing the computation speed inline with the Csound serial version. `t07` the completely parallel test features no use of read-write global variables and is not effected by their implementation. The largest gain is in `priest` which is the most computationally heavy piece. The

addition of read-write global variables change it from completely serial to mostly parallel in a similar fashion to `t09`.

5.3.6 Component Caching

Having seen the performance of the dynamic caching implementation, we were eager to see if the component design was better. We created a new stream structure using ideas from dynamic caching like originals and working variables.

We created a new method for identifying the connected graphs without using the internals of DAG nodes and sets, and minimising dynamic memory allocation. This used a 2D array representing every row as a set. When we wanted to merge sets we copied the ones from the second row into the first. We merged rows using the same concept as the non-caching version: joining sets of nodes if an edge connected them. Once we had our connected sub-graphs we merged extra rows until until their number was less or equal to the number of threads.

We used the rows as instructions to build the DAGs and then sorted them to determine the order of each stream. Again at each `kperformance` each thread would select the stream based on the thread index. Once the thread has the stream it iterates through it, performing each instrument in it.

Piece	Time (s)		
	Serial	Dynamic	Component
priest	54.11	33.68	55.01
t07	7.83	8.13	8.25
t09	10.79	11.10	12.14
xanadu	44.40	35.62	36.49
trapped	17.68	16.36	18.60

Table 5.17: Component vs. Dynamic Caching Performance Times

The cache mechanism is the same as the hash cache for the dynamic design (Section 5.3.5 on Page 41). We can see from Table 5.17 that component is slower than dynamic in all cases. Component is slower than serial Csound in all cases but `xanadu`.

5.3.7 Conclusion

We expected the component design to be faster because there is less locking during each `kperformance`. This is the critical section of the code and any attempt to reduce work here would be rewarded. These assumptions are based on the concept of having n processors with n threads running all the time, and that whenever a thread is unblocked it begins executing immediately.

This ignores the real world issues with the scheduler. Both threads may not be running at the same time. Due to the synchronisation before and after each `kperformance`, if one thread is not running all others will have to wait for that thread to return before they can continue. So if a thread with a large amount of work in its stream is pre-empted then not only do we have to wait for it to come back before we can finish the `kperformance` and find more work to do, we also have to wait for it to play its stream when it does come back. With the dynamic design if one thread is pre-empted the others can do its work in its absence. Then when it returns it only has to check that work is done before proceeding to the end synchronisation and unlocking the other threads.

This is the reason why `xanadu` was faster than serial in the component design, but `t07` was not. The overhead of the scheduling issues is masked by the improvements in parallelism in the `xanadu` case, it is not in `t07`.

It is also worth mentioning why we did not use any of the heuristics discussed in Section 5.2 (Page 26) in the dynamic design. As the sorting algorithm holds a lock on the DAG, we decided that minimising time spent inside the locked section of code was a worthwhile goal. We saw with the root countdown how reduction of time spent here yielded an improvement.

5.4 Weighting System

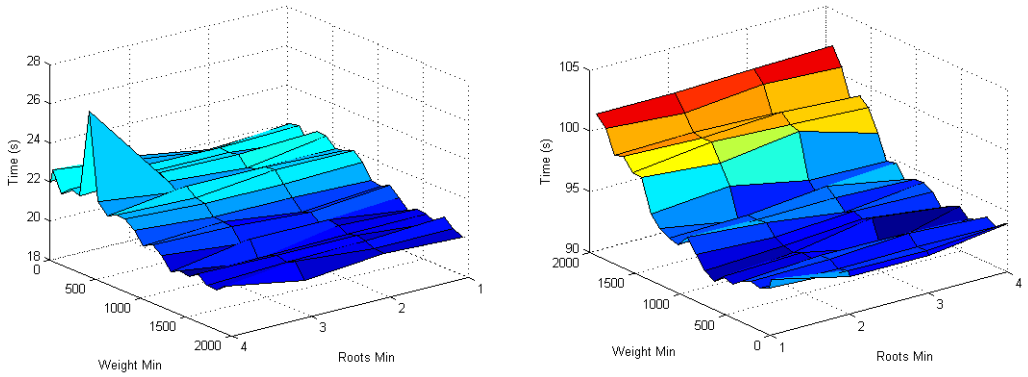
The weighting system is based on the complexity analysis ideas described in Section 2.8.2 on Page 17. We have found that parallelism does not make all programs execute faster so we need to attempt to mitigate the worst cases. For us the worst cases are pieces with light computation. We need a weighting system to give us a weight for the DAG in a `kperformance` and then decide on whether to compute this `kperformance` in parallel.

As an initial simple approach we assigned all opcodes a weight of 5 and assignments a weight of 1. This essentially serves as a count of all the operations that occur in a `kperformance`. We played each of our test pieces to determine the minimum and maximum weights for any `kperformance`.

Having developed a system of weights and found the minimum and maximum weights we wrote a testing script to determine the pivot point. It played each of our test pieces for a range of weights between the minimum and maximum, timing each one. Following this we graphed the weights versus times and looked for a minimum.

We grouped the pieces into heavy and light and summed their computation times as show in Figure 5.13 on Page 47. Unfortunately we found that the light and heavy computation pieces did not converge on a pivot weight. Heavy computation preferred lower weights as the pivot. Light computation preferred those weights greater than the maximum weight for any `kperformance`.

We needed a new, more accurate weighting system which would hopefully converge on a pivot weight. We decided to time each opcode during a performance and place the result into the weighting system. We added the ability to load and export weights for opcodes.



(a) Light Group (b) Heavy Group

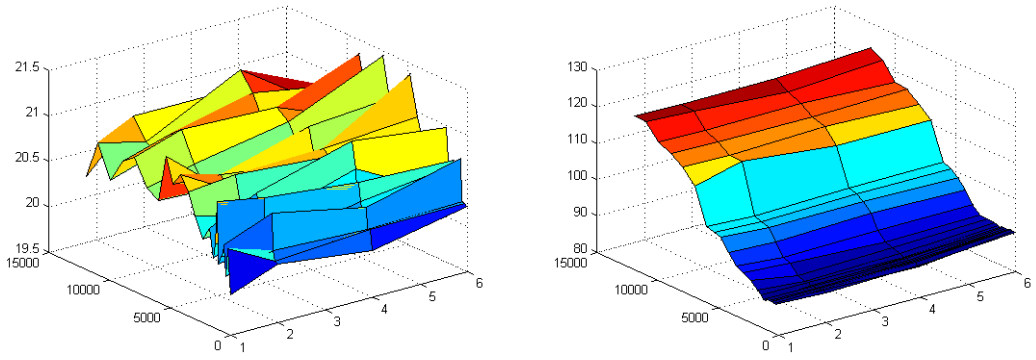
Figure 5.13: Cumulative Computation Times For Varying Weight Pivots

So we developed both a means to load weights as well as an ability to calculate them. This also meant we were able to quickly attain a subset of opcode weights necessary for our test pieces merely by running our test pieces. This saved us from having to go on a possibly very time consuming tangent just to produce a system which may not work.

Times when calculate were added into the system via a low pass filter to prevent any wide variations ($0.9 * time_{current} + 0.1 * time_{new}$). At the end the weights were calculated from the times by spreading them across a range of 1–100 inclusive. This new system was again ran through the test script and the graphs are included in Figure 5.14.

The preferred weight for the heavy group was less than 2000. The Light group also has its lowest times in the less than 2000 group. Unfortunately the best times for each piece as seen in Table 5.18 on Page 48 are no better or worse than without the weighting system. We consider two possible causes:

- When there is little work to be done in parallel in the light cases we still introduce the overhead of computing the DAGs.
- At each stage we add the overhead of a check whether to compute in parallel. The serial case is in the else clause so we may have issues with branch predictions: the processor has more work to do in the case where we want it to be faster.



(a) Light Group (b) Heavy Group

Figure 5.14: Cumulative Computation Times For Varying Weight Pivots on the 1–100 Weight Scale

Piece	Weight	Roots	Time (s)	
			Weighting System	No Weighting System
priest	1300	1	33.53	33.68
t07	3000	4	8.20	8.13
t09	100	1	11.44	11.10
xanadu	3000	6	37.04	35.62
trapped	300	1	16.54	16.36

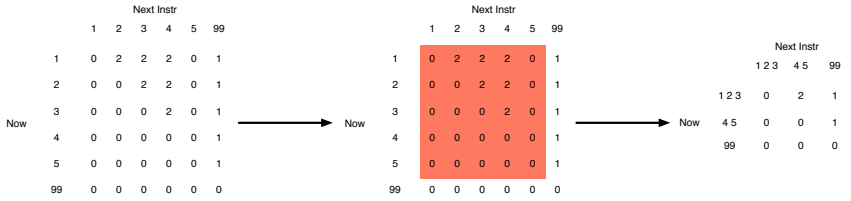
Table 5.18: Best Times for each piece with associated weight and maximum number of available roots

We knew we still had some mileage left in the component design with the introduction of read-write global variables still to do. So we leave the weighting system at this point.

5.5 Combined

We saw previously in Section 5.3.5 (Page 44) that the implementation of read-write global variables in the dynamic design had the greatest effect on the computation time. We considered the issues of implementing them in the component design. We knew we would have to have some kind of synchronisation mechanism in the streams. That way we would be able to split up previously serial DAGs to connected components followed by a reverb instrument.

We had the idea that instead of adding synchronisation to the component design, we would start with the dynamic design and introduce streams into appropriate places. That way



(a) Example 1

(b) Example 2: Rearranging instrument 3

Figure 5.15: Combined Optimisation Examples

we have our synchronisation mechanism already built. This design is a weaker form of component in that we do not find connected sub-graphs but rather groups of nodes that can be performed at the same time. We then group these nodes into streams. So now when a thread consumes a node it receives either a single instrument or a list of them to perform. This allows us to leverage the existing dependency system as our means of synchronisation. The dependencies of all the instruments in a stream are grouped and form the dependencies of the node replacing those instruments in the stream.

We can see an example of the simple case in Figure 5.15a (Page 49) where we replace five instruments that can be performed at the same time with two streams. Figure 5.15b (Page 49) shows how this design rearranges instruments to try and find the largest group that can be performed at the same time. In this case moving 3 to the end so we can perform instruments 1, 2, 4 and 5 at the same time.

So rather than a component design with read-write global variables we see this as more of a combined design. We get several advantages from this combination:

(1) If one thread is sleeping, another can take its stream, we do not have to wait for it to perform the stream when it wakes up.

(2) Streams reduce the number of calls to `consume` and `consume_update` in the light cases. This of course reduces the number of locks performed and hence reduces the number of serialisation points.

(3) We have more opportunity to try and optimise our order of instruments. In the dynamic case we cannot really perform any optimisations as we have to choose which instrument to perform in `consume`. We want to exit this function as fast as possible to

minimise time we are holding the DAG lock. We did not have the weighting system when designing component and we also did not have the infrastructure to optimise the streams.

In the combined design we use a round robin allocation of instruments to streams starting with the heaviest instrument. So if the number of instruments to put into streams modulo the number of streams is not zero, the residue end up in the lower slots. We place lower slots earlier in the DAG. Essentially we make the thread that fetches the first stream have the most work. This is good as that thread is clearly running at the time so we do not end up with a situation where a sleeping thread wakes late and takes a steam with the most work delaying everyone else.

We implemented this design and the times for the versions of Csound are contained in Table 5.19 and Figure 5.16. We can see improvements to the light computation pieces which put their times at a sub-serial level. We also see a dramatic improvement to `xanadu`. `priest` is slightly worse than the dynamic design, but still vastly better than the original serial. `trapped` is not much better than the original serial.

Piece	Time (s)			
	Serial	Dynamic	Component	Combined
priest	54.11	33.68	55.01	34.60
t07	7.83	8.13	8.25	6.46
t09	10.79	11.10	12.14	9.28
xanadu	44.40	35.62	36.49	27.29
trapped	17.68	16.36	18.60	16.29

Table 5.19: Combined Performance Times

The improvements to `xanadu` are due to its lack of any global variables, so combined implementation functions here like the component design. Reasons 1 and 3 from above have the most impact here. This piece is quite heavy so reason 2 has less of an effect.

The improvements to `t07` arise from similar issues, it has no global variables and so we see the reduced overhead from the reduction of all the DAGs to 2 streams. The reason it performs better than component is probably caused by all the reasons discussed above.

`t09` is the same as `t07` but with a global reverb at the end. The reasons for improvements are the same as `t07` with the caveat that it still has to perform the reverb after the streams.

5.6 More Than Two Processors

We had the opportunity to test the combined design on a four processor machine[1]. The machine runs Windows XP with the MinGW tool chain. This is quite different to our

[1] a 2.86 Ghz AMD Phenon X4

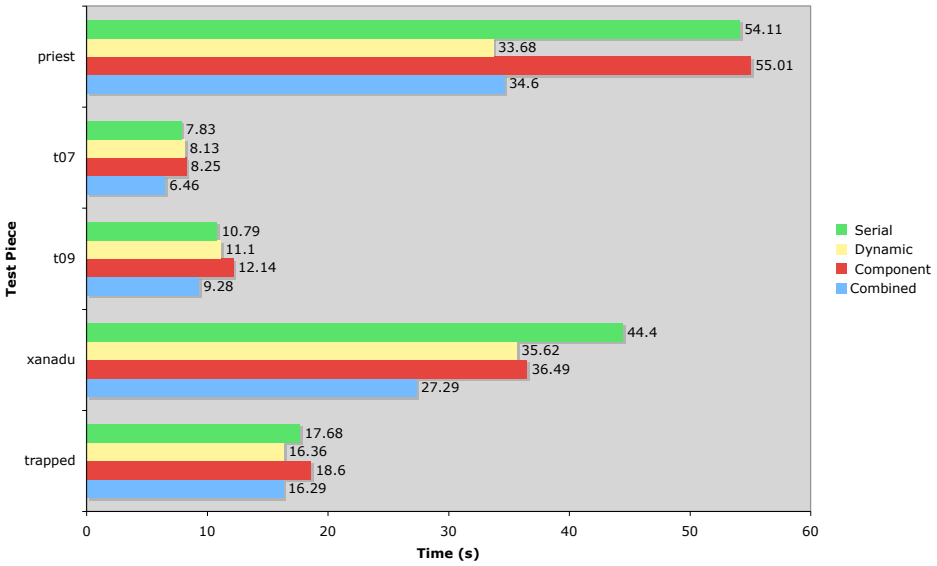


Figure 5.16: Times for each piece for each Design

testing setup, which we discussed in Section 4. We performed our standard suite of tests in serial and with 2–4 threads and the results are in Table 5.20.

Piece	Time (s)					
		4 Processors			2 Processors	
	Serial	2 Threads	3 Threads	4 Threads	1 Thread	2 Threads
priest	46.40	32.59	28.34	29.41	47.10	32.76
t07	6.54	5.93	7.38	8.88	6.40	6.31
t09	8.72	9.86	11.93	13.48	8.76	10.18
xanadu	33.85	25.28	31.77	44.35	34.43	26.92
trapped	16.98	17.49	20.64	25.69	16.97	17.67

Table 5.20: 4 Processors and 2 Processors Performance Times

The time improvements are not as clear cut as on our test machine. There are improvements to `priest`, `t07` and `xanadu` with varying numbers of threads over the serial time. `t07` and `xanadu` are faster with two threads, while `priest` is faster with three. `t09` and `trapped` show no improvement.

`t07` and `xanadu` benefit most from the DAG optimisation in the combined design. We have seen for our test pieces that the optimisation reduces the DAG to a number of nodes—containing streams—equal to the number of threads, followed by reverb instrument nodes. As the number of threads goes up the number of nodes in a DAG to be consumed goes

up. Consuming occurs in serial and is part of our parallelism management overhead. So as we increase the number of threads we have more overhead compared to a lesser number of threads. This can explain the `t07` results where the overhead for managing parallelism exceeds the benefits at the three thread point. This does not explain the result of `xanadu` with four threads. We would expect that for a piece of this length the caching amortises the DAG optimisation over the length of the piece. We would never expect it to exceed the time taken in serial.

We needed to investigate this, but first we discuss the issue of running with a number of threads less than the number of processors. When we run in this configuration we expect that the scheduler will be less likely to pre-empt our threads. When we run with all processors the scheduler needs to pre-empt so it can do work itself, and also to let other operating system work occur and to some extent any other processes which may be running. When Csound is not using all the processors the scheduler can use one to do the aforementioned work while mostly letting Csound do its work. This is why running with two threads on a four processor machine is not comparable to running with two threads on a two processor machine in our testing. If the machines were special purpose and only running our program then this comparison may have been possible. Fortunately there is a `msconfig` setting that allows starting Windows with less than the total number of processors on the host's hardware.

We tested with just two of the four processors enabled so we could compare the results with our test machine. The results are contained in Table 5.20. We see mostly equivalent performance gains, although the four processor machine is faster in serial than our test machine. Again however `t09` and `trapped` perform worse, where we would expect performance gains.

Our initial setup on the four processor machine uses gcc 4.2 so we have the atomic instruction (`__sync_lock_test_and_set`) to use for our spinlocks as discussed in Section 4. We do not have an equivalent to the `OSAtomic` Mac OS spinlocks. We investigated whether this could be the cause. Table 5.21 contrasts the results of a test (back on our test machine) of the combined design Csound compiled with gcc 4.0 and `OSAtomic` spinlocks against gcc 4.2 and `__sync_lock_test_and_set` based spinlocks.

Piece	Time (s)		
	gcc 4.0	gcc 4.2 & `__sync_lock_test_and_set`	gcc 4.2 & `OSAtomic`
priest	34.60	34.16	33.87
t07	6.46	7.05	6.35
t09	9.28	9.61	9.09
xanadu	27.29	27.18	27.29
trapped	16.29	17.12	16.78

Table 5.21: gcc 4.0 with OS Atomic vs. gcc 4.2 using `__sync_lock_test_and_set` Performance Times

We see that gcc 4.2 with the `__sync_lock_test_and_set` based spinlock performs worse

than with the `OSAtomic` version. This is probably one of the reasons for some reduced performance of `t09` and `trapped` when used on the four processor machine with just two processors enabled. This will of course affect all the results to some extent that we have seen from the four processor machine. This performance difference is likely due to the use of a backoff used in the `OSAtomic` spinlock (Apple Inc., 2004). When between `kperformances` and the worker threads are spinning on the synchronisation barrier the thread will backoff and perhaps sleep for a short time. This may have some effect on the internal measures the scheduler uses to determine whether to pre-empt that thread. Essentially by spinning doing no work we make it more likely we will be pre-empted when we want to actually do some work.

With these issues we cannot really draw any conclusions about performance of our combined design on more than two cores with any certainty. With a backoff version of the Csound spinlock we would have some more comparable results. We do see that while the weighting system was not useful for determining whether to compute a DAG in parallel, it could be used to determine the best number of threads to compute a piece with.

5.7 Real-Time Csound

All of our test pieces are computable in real-time in serial on our machine. We crafted a simple altered version of `priest` where the score was performed three times, but the start was offset by half a second. This roughly triples the work in `priest` which is already the most computationally heavy test piece. We calculated the non-real-time rendering time first to demonstrate the difference between serial and parallel. The results are contained in Table 5.22.

Piece	Time (s)	
	Serial	Parallel
priest altered	151.46	91.69

Table 5.22: Real-Time Performance of Priest Altered in Serial and Parallel

We then performed each piece in real-time and listened for any errors in the output. The serial version performed fine up to 70 seconds where the output became noisy and jittery. For the parallel version we detected no audible errors up to 115 seconds. However at this point in both pieces there were a large number of samples out of range. So past 115 seconds we cannot tell whether it is the clicking caused by the samples out of range or that part of the piece is not computable in real-time. The cause of the samples out of range is because we have tripled the number of instruments playing without decreasing the amplitude of their output.

Our parallel implementation allowed us to perform a non real-time computable—in serial—contrived test piece in real-time. We would like find a real example of a piece that was non real-time computable to discover the limit to the extra work that can be done in real-time.

5.8 Compiler Optimisation

All our tests up to this point have been compiled with gcc 4.0 and no optimisation—with the exception of the four processor tests—as discussed in the Test Plan (Section 4). We enabled optimisation and the results for the test pieces in serial and parallel are contained in Table 5.23. We continue to use gcc 4.0 and compile with the flags: `-ffast-math`, `-ftree-vectorize`, `-O3` and `-mtune=generic`. This version of gcc does not support compiling for our specific processor type so we leave the `-mtune` option as generic.

Piece	Time (s)	
	Serial	Parallel
priest	25.81	18.20
t07	4.60	4.72
t09	6.11	6.60
xanadu	27.27	18.71
trapped	11.20	12.02

Table 5.23: Serial vs. Parallel with Compiler Optimisation Enabled Performance Times

From the results we see that parallel Csound is not always faster than serial. `t07` and `t09` are slightly slower in parallel. As discussed previously these light computation pieces are on the edge of the management of parallelism versus parallelism gains ratio. So when we reduce the work done in parallel, we reduce the benefits we gain from it. The overhead will be reduced as well but probably not as much. This is because operations which previously took a long time like copying a buffer to output will be speeded up more by the vector instructions than removing a few instructions from finding a root to play in `consume`.

`trapped` is another case that performed better in parallel but not by much. The reasons for this performance reduction are similar to those of the light computation pieces `t07` and `t09`.

`priest` and `xanadu` both show the usual performance gains.

Chapter 6

Conclusion

We first provide a brief overview of the steps taken in achieving a performance improvement and the conclusions made therein. We then look at some other areas of the implementation and the results of the testing. We sum up our overall conclusions, make recommendations for a theoretical future version of the project and finally enumerate the future work.

6.1 Performance

Performance improvements only occurred when we found parallelism, and the overhead of managing that parallelism is less than the benefit of it.

As an example in the dynamic design as soon as we had the cache, `xanadu`—which is completely parallel—saw a speed up. Serial pieces like `trapped` and `priest` saw no speed up, just the effects of the overhead added to manage parallelism. In the case of `t07` the overhead of managing parallelism was greater than the benefit despite the fact that it is completely parallel. We see a similar effect for `xanadu` with the component design where the overhead of managing parallelism was lower than the benefits.

To unlock further parallelism in the dynamic design we added the concept of spinlocks around read-write global variables so instruments dependent on them could be performed at the same time. This change makes all the pieces parallel up until the reverb (and delay in `trapped`) instrument at the end. After this we see the large improvement in `priest`. We also see a benefit in `t09` the light computation (previously) serial piece, which is now almost fully parallel. For this piece though, the overhead of parallelism is still greater than the benefits.

We saw that the problem was one of deciding when the overhead would exceed the benefits. If we could decide before a `kperformance` we would always choose the faster option: serial or parallel. This did not take into account a factor of the overhead was the building of DAGs: a necessary part in determining the benefits of parallelism. We also added a decision at every `kperformance` which—with the branching concerns—can add up when the we have

a `kperformance` 4410 times for each second of a piece of music.

We were resigned at this point to the idea that we would be unable to get any further improvements from the system. A conclusion in mind, that we would only be able to show performance gains once the work was sufficient, overriding the synchronisation and other overheads.

We looked back at the component design and saw respectable times in the caching version. At the time we had been focused on the light computation pieces `t07` and `t09`. We were unable to get them to sub-serial times even with attempts to keep the threads running as much as possible. Together with the realisation that the read-write global variables was the largest factor in the dynamic speedup, we had to implement them in the component design. We decided that in fact a hybrid would make the best use of both designs as it already had the synchronisation we would require.

We considered how to implement the finding of connected components when we essentially had a fully connected graph. We realised that what we wanted was to group those instruments which could be performed at the same time. We split these groups and put the streams into new DAG nodes capable of holding them. We used the opportunity of doing this at DAG creation time to optimise the streams spreading out the work based on weight with a bias to the main thread.

With the implementation of the combined design we see a reduction in the overhead of managing parallelism. When a thread receives a stream of three instruments, it is 1 call each to `consume` and `consume_update` instead of 3 each. This not only reduces the number of function calls and locks it also reduces the time that other threads cannot access the DAG. This makes it more likely that when another thread wants to do some work it will not have to wait for access to the DAG.

6.2 Semantic Analysis

The implementation of semantic analysis was fairly straightforward, certainly more so than the dispatching element of the project. This was probably because of the lack of concerns over efficiency: the analysis only occurring once at the the start of the computation of a piece.

The decision to use a mixture of an AST walk and the parser worked well. However in light of the later addition of the read-write global variable lock inserts and the weight calculation tree walks it would be more consistent to implement the semantic analysis fully as a tree walk.

We have some concerns about the fullness of the analysis. Global variable usage in conditional statements is not checked, neither is type of operation in a read-write global variable operation. There is obviously some future work to be done here.

6.3 Accuracy

At each stage the accuracy of the pieces was tested sample for sample against a canonical version produced by serial Csound. All of output is done in signed 16-bit WAV format. We noticed that the largest difference is 1 for any sample.

Most of the computation in Csound is done with floating point, so a change in the order of computation can effect the result. We would expect that this would produce errors sufficiently small that they disappear with the information loss when rounding to signed 16-bit integers. We see that these 1 off samples do not occur all the time, so it is possible that for these occurrences the floating point value was very close to a rounding boundary. So in two different runs depending on the order the operations on the global variable the floating point value could fall either side of the boundary.

The number of these occurrences increases with the number of opportunities for global variable calculations to be rearranged. So the problem is at its worst when read-write global variables are used. We cannot remove read-write global variables as they are the single biggest factor in the parallel speed up.

We could remove the issue by calculating the intermediaries up to interaction with the global variable. These could then be computed in serial order when all the intermediaries across all instruments were known. At the end of the read-write group of instruments before the instrument that performs a read or write to the global we perform the computation. Essentially we use lazy evaluation.

A similar issue occurs with the writing to the output with the `out` family of opcodes. These add the buffer of samples given as an argument into the output buffer. Again we can see that the order in which this is performed can effect the output as these values are floating point. This could be solved in a similar manner to above where `out` writes to a buffer for that instrument, at the end of the `kperformance` all the instrument output would then be written in serial order to the main output buffer.

Obviously these two solutions could dull the parallel speed ups. Particularly the `out` issue as this would be done after the `kperformance` when the program was running in serial. The benefits would definitely be worth it to restore the deterministic nature of serial Csound, that a piece computed twice will give the same output.

By letting this issue stand we have failed to meet Non-Functional Requirement 1.3 (Page 21). We consider that the error would not be detectable to human ears, which mitigates this somewhat. This is also an area for future work.

We have included the percentage of samples that differed from a reference serial output in Table 6.1. The serious error category is for those where the difference is greater than 1. The number of incorrect samples is very small.

Piece	Error (%)	Serious Error (%)
priest	0.236	0
t07	0.079	0
t09	0.079	0
xanadu	0.059	0
trapped	0.067	0

Table 6.1: Percentage Error For Test Pieces

6.4 More than Two Processors

We saw in Section 5.6 that the speed up did not continue obviously onto greater numbers of processors than the two in our test machine. There were some issues surrounding the testing which prevent us from drawing a firm conclusion on this: the different compiler, spinlock and operating system. However we do foresee the issues we have faced in attempting to gain a speedup for two processors being magnified with more.

- The wait to proceed past synchronisation when one thread has been pre-empted and is sleeping. With more processors and more threads there is a higher probability of one thread not being awake when needed.
- Parallelism overhead management. With more threads and processors our DAG optimisation puts less work into each individual consume chunk. So there are more consumes overall increasing the serial proportion of the program. Each individual thread has to do more work as a call to `consume` is a greater piece of work than fetching the next instrument from an array.

It is possible in some situations for these problems to be reduced, but it is very unlikely.

There is also an upper bound on the parallelism available. We split instruments across threads, if in a piece only one instance of any instrument is playing at the same time there is no parallelism available. We do not think this is representative of Csound pieces, but we should consider the upper bound on our test pieces. If a piece has a maximum of 18 instruments playing at the same time, then our upper bound on parallelism is at 18. We do not have a suggestion at this time of a way to increase parallelism beyond this.

Piece	Number
priest	18
t07	32
t09	33
xanadu	24
trapped	14

Table 6.2: Maximum Number of Instruments Playing at Once

The actual maximum number of instruments playing at one time are contained in Table 6.2. These are all fairly low numbers when considering many-core type machines, especially considering the more complex pieces have lower upper bounds on parallelism.

6.5 Conclusion

We will now examine whether we have met the aims we set out to achieve and summarise the results of our work.

6.5.1 Evaluation of Requirements

In determining whether we have met our aims we examine our list of requirements from Section 3 and evaluate how well we have satisfied them.

Satisfaction of Functional Requirements

(1) We implemented the semantic analysis system to determine dependencies of instruments on global variables.

  (1.1) The semantic analysis system works for numbered instruments, but does not support named instruments.

  (1.2) We do not recognise all use of global variables: we do not recognise those in conditionals. None of our test pieces use them, but this should be implemented for completeness.

  (1.3) We support the orchestras in the test pieces but we have not tested on a greater range. The test pieces are representative of Csound pieces and we think a reasonably range given the time scale of the project.

(2) We use the semantic analysis system from Requirement 1 to determine the dependencies of an instrument on other instruments.

(3) We have implemented a parallelism mechanism which computes instruments in parallel.

  (3.1) The mechanism is built with Pthreads.

  (3.2) We have only tested on Mac OS X and Windows with the MinGW tool chain. We have not tested the portability any more widely than this. We have endeavoured to ensure there are no warnings and none of the code relies on any platform specific features so hopefully this will not be an issue.

(4) We provide the weighting system which can specify the weight of an opcode.

(5) We provide a combination of the weighting system and a custom build of Csound to calculate weights of opcodes in a crude time based fashion.

Satisfaction of Non-Functional Requirements

(1) The semantics of the input program are mostly preserved, with the exception of the floating point issues discussed in Section 6.3.

  (1.1) Global variable access occurs in a mostly semantically equivalent fashion, again excepting the floating point issues.

    (1.1.1) We cannot perform this rearrangement without effecting the result of the computation. This requirement was very optimistic and probably un-satisfiable. On the other hand without it we would have not seen the performance gains we achieved. We do however preserve the global value between reading it and writing the new value back in.

    (1.1.2) We have not implemented this requirement. This would require us to determine whether a global variable was no longer used after the blind writes. If it is used we must preserve the order so the next reader finds the expected value

    (1.1.3) Multiple reads do occur simultaneously.

  (1.2) Access to shared output was already protected with a spinlock. We do again have the floating point calculation reordering issue as with global variables.

  (1.3) We do verify sample by sample and as discussed in Section 6.3 the largest difference for any one sample is one, where samples are signed 16-bit integers. So we do not meet this requirement, but a difference of one is probably inaudible anyway.

(2) When run in serial the program is only takes the extra time to perform semantic analysis. Other operations are only performed if parallelism is specified by the user.

(3) For our test machine the performance is improved when working in parallel. We have no test pieces which are entirely serial with the combined design. If we did it is likely they would perform worse based on our observations about parallelism management overhead versus parallelism benefits. We have not tested with an entirely serial piece as we do not consider one to be representative of a typical Csound program.

  (3.1) Our optimisation in the combined design fulfils this requirement that we take advantage of the available number of processors while taking into account communication. We saw that there is a possibility that more will need to be done with more than two processor machines to determine whether to use less threads than processors.

  (3.2) We attempted with the weighting system, to determine whether to perform a task in parallel only if a performance gain would result. We found the overhead of that determination caused a performance drop. We have met this requirement but not the spirit of it: we do everything in parallel and overall a performance gain is the result.

(4) The process of parallelising the program is invisible to the programmer aside from specifying a number of threads to use.

(4.1) New opcode were added to the Csound language, but these are just for internal use.

(4.2) The programmer does not have to give any 'hints' about where to parallelise.

Satisfaction of Requirements Summary

We have met the requirements relating to speed and met a weaker form of the semantics preserving ones. In Future Work (Section 6.7) we discuss designs which could be used to meet the requirements we have missed. In effect we have met the spirit of the goals of the project but there is remaining work to be done to tidy up the edges.

6.5.2 Results Summary

All of our performance gains come from increasing the amount of parallelism whilst reducing the overhead for managing parallelism. We also had to take into account the realities of a parallel program running on a non-specialist machine, that it is often interrupted.

In doing this we endeavoured to reduce the number and length of serialisation points where multiple threads accessed the DAG structure. We reduced the length of the serialisation inside the `consume` / `consume_update` functions with the root countdown. We reduced the number with the DAG optimisation which introduced streams so a thread could fetch several instruments to perform in a group.

We worked to increase the amount of parallelism available in the DAG with the introduction of read-write global variables. These allowed all of the serial test pieces to be computed mostly in parallel.

We have validated both Amdahl's Law and Gustav's Law (Section 2.3 on Page 6). We have seen that we can do more work in the same time than we are currently doing in serial. We can see that we can only reduce the time taken down to the time taken for:

- the serial part of the program plus
- the longest piece of parallel work plus
- the overhead of managing parallelism.

In Section 2.3 we suggested that making a non real-time computable piece computable in real-time was a real gain. Subject to the conditions we specify in Section 5.7 we have achieved this.

It is unlikely that we have found the boundary on the minimum amount of time taken to compute our test pieces, however there will probably be a diminishing returns for further

effort put in. We suggest numerous improvements that can be made in the Future Work Section (Section 6.7 on Page 63). Of course some of the improvements particularly related to floating point and read-write global variable work may reduce some of the speedup.

6.6 Reflection

In hindsight there were of course things that should have been done differently. We detail these issues and possible solutions.

Begin writing up earlier. Writing this document focused our attention on the causes of the speedup. It also forced us to look more closely at the results and prompted the implementation of the combined approach. Before this we were content with the conclusion that we had found the limit in lowering our overhead, and that we should just mitigate the damage caused by the overhead on light computation pieces.

Focus on all the test pieces sooner. We only realised with more widespread testing a bug which had persisted for some time. We thought the implementation of the counting semaphore was the fix for an architectural issue, actually it was a fix for this bug. We saw after fixing the bug that the counting semaphore reduced performance. Not really knowing enough about Csound may have caused the non-diverse testing. We also focused far too much on attempting to make those light pieces faster with the fallacy that improvements to them would surely lead to improvements in other pieces.

A realisation that rearranging floating point operations is not necessarily semantically equivalent. We could then, have at least implemented a prototype of the fixes discussed in the Accuracy Section 6.3 (Page 57). We were aware of these issues but as mentioned we thought the information loss during the conversion to 16-bit signed integers would hide the slight errors.

Better test structure. We had some unit tests for the early implementation, but did not attempt to create them for the dispatching. This would have prompted us to better understand the code around the dispatching. It would also make testing a wider variety of DAGs easier.

Better use of source control. We implemented up to dynamic caching on a single branch using hash defines to control which one to use. We faced a loss of time when trying to fix a bug amongst that mess. Hash defines are useful to switch various options for performance testing but usually only after implementation, testing and debugging. We used branches after dynamic caching and found management of the multiple designs much simpler.

We left some bugs around for far to long, particularly the thread id issue which was caused by an existing race condition. This made it very difficult to conduct automated testing as the symptom was one thread would exit and Csound would hang at the next barrier. We partially fixed the issue early on so it would at least exit. This made automated testing possible, but complicated by return code checking.

6.7 Future Work

There are numerous task still to be done. This is particularly true with the later work which took on more of a prototype implementation status.

(1) Named instruments. Csound supports named instruments, we need to find where these fit into the order of played instruments and work them in. This may present issues with the mechanism used to identify the right DAG from the cache, which relies on a list of integers.

(2) Semantic Analysis improvements. We discussed in Section 6.2 on Page 56 the issues present with semantic analysis. We need to add support for writing to tables, proper support for conditional statements and all the other occurrences of global variables in the language. Finally we need to ensure that read-write statements can only increment or decrement a global variable, we cannot allow `*=` or `/=`.

(3) Floating Point. We outlined in Section 6.3 on Page 57 the issues with read-write global variables and the `out` family of opcodes.

(4) Multiprocessor testing. We need to test on a machine with more than four processors to find out where the performance drops off.

(5) Investigate Component Parallelism. Is it worth attempting to introduce a connected component optimisation to the combined design. In that if we have a piece like our example orchestra (Listing 5.1 on Page 25) it performs like the component design on a dual core machine. Rather than performing the instruments 1, 2, 3, and 4 in streams and then consuming 5 and 6 separately. Does this provide a speedup and are there sufficient orchestras with this kind of instrument pattern to make it worthwhile?

(6) Weight investigation. When working on the combined design we noticed only a small difference between using weights calculated from timing opcodes in the weighting system and using the five for an opcode, one for an assign system. Do we really need accurate weights or is a simple count of opcodes in an instrument sufficient?

(7) Orchestra investigation. We need to test a greater range of orchestra and compare them sample to sample with the serial Csound. This would be much simpler if done after the work on the floating point issues. The output should then be sample accurate and this testing could be fully automatic.

(8) Automatic number of threads. We need to introduce a portable function to determine the number of processors a users machine has so we do not have to ask the user for the number of threads to use. Of course some people on more than two core machine may prefer this so they get a speed boost but it does not take up all their available processors.

(9) Processor usage. Following the last point we know that while not in a `kperformance` the worker threads are wastefully blocked. Is there a way we can sleep them while still having them responsive so they are ready to begin work as soon as possible. Perhaps we can have the first barrier before work as a mutex barrier allowing worker threads to sleep while the main thread decides what instruments to play. The second barrier would still use spinlocks so when we finish a `kperformance` the main thread can go back to working out what to play next as fast as possible.

(10) Different parallelism model. Perhaps with more knowledge of the other areas of Csound we could change the model such that the main thread determines what to do for the next `kperformance` at the same time as a worker thread is computing the current `performance`.

(11) Csound integration. We have only integrated with a narrow code path through Csound. There will be extra work to do to support the API with potential issues like parsing multiple files. Here the instrument semantics structures and the caches of DAGs will have to be deleted at the appropriate places. There are probably other unforeseen issues.

(12) Guard `globallock` / `globalunlock`. These functions are implemented as opcodes, so a programmer could put them in their program. There is very little opportunity to use these correctly. It would be worth detecting the prior existence of these opcodes when inserting them in the read-write global variable lock insertion AST walk. If we find the programmer has used them we should report an error and exit.

(13) buildNewParser hash define. The build system uses the option buildNewParser to determine whether to use the YACC / Bison based parser. We need to properly integrate our work with this so it is only compiled with this option set.

(14) Potability issues. We saw a drop in performance with the standard Csound spinlock. We need to implement something similar to the `OSAtomic` Mac OS spinlocks to fix this. There may be other issues lurking that will appear once we have eliminated this.

(15) Bugs. As a matter of record we occasionally see a bug where a DAG is accessed by a worker thread after the piece has been performed. This is hard to reproduce and requires further investigation.

With just the introduction of the floating point operation order protection items we have a fully working system which provides real performance gains. These gains are felt by both people rendering pieces to files, and those doing real-time performances.

Bibliography

Amdahl, G. M. (2000), *Validity of the single processor approach to achieving large scale computing capabilities*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Andrews, J. and Baker, N. (2006), 'Xbox 360 system architecture', *Micro, IEEE* **26**(2), 25–37.

Apple Inc. (2004), 'Mac OS X manual page for spinlock(3)', `http://developer.apple.com/documentation/Darwin/Reference/ManPages/man3/spinlock.3.html`. Last Updated.

Apple Inc. (2008*a*), 'About Mac OS X threads', `http://developer.apple.com/documentation/Cocoa/Conceptual/Multithreading/CreatingThreads/chapter_4_section_2.html`. Last Updated.

Apple Inc. (2008*b*), 'Shark User Guide: Introduction', `http://developer.apple.com/documentation/DeveloperTools/Conceptual/SharkUserGuide/Introduction/Introduction.html`. Last Updated.

Carriero, N. and Gelernter, D. (1989), 'Linda in context', *Commun. ACM* **32**(4), 444–458.

Fitch, J. B. M. . J. P. (1983), The Bath concurrent LISP machine, *in* 'EUROCAL '83: Proceedings of the European Computer Algebra Conference on Computer Algebra', Springer-Verlag, London, UK, pp. 78–90.

Fitch, J. P. (1988), A loosely coupled parallel LISP execution system, *in* 'The Design and Application of Parallel Digital Processors', Vol. 298 of *IEE Conference Publication*, IEE, pp. 128–133.

Fitch, J. P. (1989), Can REDUCE be run in parallel?, *in* 'Proceedings of ISSAC89, Portland, Oregon', SIGSAM, ACM, pp. 155–162.

Flynn, M. (1972), 'Some computer organizations and their effectiveness', *IEEE Transactions on Computers* **C-21**(9), 948–960.

Geer, D. (2005), 'Chip makers turn to multicore processors', *Computer* **38**(5), 11–13.

Gibbons, P. B. and Muchnick, S. S. (1986), Efficient instruction scheduling for a pipelined architecture, *in* 'SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction', ACM, New York, NY, USA, pp. 11–16.

Gross, T., OHallaron, D. and Subhlok, J. (1994), 'Task parallelism in a high performance fortran framework', *Parallel Distributed Technology: Systems Applications, IEEE [see also IEEE Concurrency]* **2**(3), 16–.

Gustafson, J. L. (1988), 'Reevaluating Amdahl's law', *Commun. ACM* **31**(5), 532–533.

Hochstein, L., Carver, J., Shull, F., Asgari, S. and Basili, V. (2005), Parallel programmer productivity: A case study of novice parallel programmers, *in* 'SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing', IEEE Computer Society, Washington, DC, USA, p. 35.

Hofstee, H. (2005), 'Power efficient processor architecture and the cell processor', *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on* pp. 258–262.

Lee, E., Ho, W.-H., Goei, E., Bier, J. and Bhattacharyya, S. (1989), 'Gabriel: a design environment for dsp', *Acoustics, Speech and Signal Processing, IEEE Transactions on* **37**(11), 1751–1762.

Lewis, T. G. (1997), *Foundations of Parallel Programming: A Machine-Independent Approach*, IEEE Computer Society Press, Los Alamitos, CA, USA.

Loveman, D. (1993), 'High performance fortran', *Parallel Distributed Technology: Systems Applications, IEEE [see also IEEE Concurrency]* **1**(1), 25–42.

Marti, J. B. (1980), Compilation techniques for a control-flow concurrent lisp system, *in* 'LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming', ACM, New York, NY, USA, pp. 203–207.

Marti, J. B. (1981), Detection of available concurrency in lisp programs, Technical report, Department of Computer and Information Science, The University of Oregon.

Marti, J. P. F. . J. B. (1987), The static estimation of run time, Technical Report 89-18, School of Mathematical Sciences Computer Group, University of Bath.

Mes (2008), *MPI: A Message-Passing Interface Standard Version 2.1.*

O'Hallaron, D. (1991), 'The assign parallel program generator', *Distributed Memory Computing Conference, 1991. Proceedings., The Sixth* pp. 178–185.

Raman, S., Pentkovski, V. and Keshava, J. (2000), 'Implementing streaming simd extensions on the pentium iii processor', *Micro, IEEE* **20**(4), 47–57.

Rau, B. R. and Fisher, J. A. (1993), 'Instruction-level parallel processing: History, overview, and perspective', *The Journal of Supercomputing* **7**(1-2), 9–50.

Rinard, M. C., Scales, D. J. and Lam, M. S. (1993), 'Jade: A high-level, machine-independent language for parallel programming', *Computer* **26**(6), 28–38.

Seebach, P. (2005), 'Unrolling AltiVec, part 1: Introducing the PowerPC SIMD unit', `http://www.ibm.com/developerworks/library/pa-unrollav1/`.

Subhlok, J., Stichnoth, J. M., O'Hallaron, D. R. and Gross, T. (1993), 'Exploiting task and data parallelism on a multicomputer', *SIGPLAN Not.* **28**(7), 13–22.

Whiting, P. G. and Pascoe, R. S. V. (1994), 'A history of data-flow languages', *IEEE Ann. Hist. Comput.* **16**(4), 38–59.

Wolfe, A. (2004), 'Intel clears up post-tejas confusion', `http://www.crn.com/it-channel/18842588`. Last Updated.

Appendix A

Code

We now present the combined design source code. Source for other versions is included on the accompanying disk. We provide both the git[1] repository as well as snapshots of the final states of each branch. The source here has been mangled somewhat to fit onto the paper, the source on the disk is in the correct format.

On the accompanying disk are the following directories:

dynamic_component the non-caching implementations of dynamic and component. This also contains the first implementation of dynamic with caching.

dynamic the final version of the dynamic with caching design.

component the final version of the component with caching design.

combined the combined design. This is the final design and the outcome of the project.

src_control a copy of our git repository so readers may browse it if they wish.

tests the test pieces we use and the special `priest_altered` used for real-time testing.

cs_par is the prefix of all the new Csound parallelism code. cs_par_base contains data structures and functions that are used both in the semantic analysis and the dispatching. It also contains all the hash define switches. cs_par_orc_semantic_analysis contains all the semantic analysis code. cs_par_dispatch contains the parallelism implementation and the weighting system.

A.1 Header Files

A.1.1 File: cs_par_base.h

[1] `http://git-scm.com/`

```
1 #ifndef __CS_PAR_BASE_H__
2 #define __CS_PAR_BASE_H__
3
4 #define DYNAMIC_2_SERIALIZE_PAR
5
6 #define TRACE 0
7 /* #define TIMING */
8
9 /* #define SPINLOCK_BARRIER */
10 #define SPINLOCK_2_BARRIER
11
12 /* #define NUM_THREADS_OLD_DEF */
13
14 #define HASH_CACHE
15 /* #define HYBRID_HASH_CACHE */
16 /* #define LINEAR_CACHE */
17
18 #define COUNTING_SEMAPHORE
19
20 /* #define CACLULATE_WEIGHTS_BUILD */
21 #define LOOKUP_WEIGHTS
22
23 #ifdef TIMING
24   #define TIMER_INIT(val, name) RTCLOCK val ## _timer;
25   #define TIMER_START(val, name) \
26     csound->InitTimerStruct(& val ## _timer); \
27     csound->Message(csound, name "Start: %f\n", csound->GetRealTime(& val ## _timer));
28   #define TIMER_END(val, name) \
29      csound->Message(csound, name "End: %f\n", csound->GetRealTime(& val ## _timer));
30
31   #define TIMER_T_START(val, index, name) \
32     csound->InitTimerStruct(& val ## _timer); \
33     csound->Message(csound, "[%i] " name "Start: %f\n", index, \
34     csound->GetRealTime(& val ## _timer));
35   #define TIMER_T_END(val, index, name) \
36     csound->Message(csound, "[%i] " name "End: %f\n", index, \
37     csound->GetRealTime(& val ## _timer));
38 #else
39   #define TIMER_INIT(val, name)
40   #define TIMER_START(val, name)
41   #define TIMER_END(val, name)
42   #define TIMER_T_START(val, index, name)
43   #define TIMER_T_END(val, index, name)
44 #endif
45
46 #define TRACE_0(...) csound->Message(csound, __VA_ARGS__)
47 #if TRACE > 0
48     #define TRACE_1(...) csound->Message(csound, __VA_ARGS__)
49 #else
50     #define TRACE_1(...)
51 #endif
52 #if TRACE > 1
53     #define TRACE_2(...) csound->Message(csound, __VA_ARGS__)
54 #else
```

```
55      #define TRACE_2(...)
56  #endif
57  #if TRACE > 2
58      #define TRACE_3(...) csound->Message(csound, __VA_ARGS__)
59  #else
60      #define TRACE_3(...)
61  #endif
62  #if TRACE > 3
63      #define TRACE_4(...) csound->Message(csound, __VA_ARGS__)
64  #else
65      #define TRACE_4(...)
66  #endif
67
68  /* #define SHARK_SYMBOLS */
69
70  #define KPERF_SYM 0x31
71  #define BARRIER_1_WAIT_SYM 0x32
72  #define BARRIER_2_WAIT_SYM 0x33
73  #ifdef SHARK_SYMBOLS
74      #include <sys/syscall.h>
75      #include <sys/kdebug.h>
76
77      #define SHARK_SIGNPOST(sym) \
78          syscall(SYS_kdebug_trace, \
79          APPSDBG_CODE(DBG_MACH_CHUD, sym) | DBG_FUNC_NONE, 0,0,0,0)
80  #else
81      #define SHARK_SIGNPOST(sym)
82  #endif
83
84  /* return thread index of caller */
85  int csp_thread_index_get(CSOUND *csound);
86
87  /* structure headers */
88  #define HDR_LEN                 4
89  #define INSTR_WEIGHT_INFO_HDR   "IWI"
90  #define INSTR_SEMANTICS_HDR     "SEM"
91  #define SET_ELEMENT_HDR         "STE"
92  #define SET_HDR                 "SET"
93  #define DAG_2_HDR               "DG2"
94  #define DAG_NODE_2_HDR          "DN2"
95  #define SEMAPHORE_HDR           "SPS"
96  #define GLOBAL_VAR_LOCK_HDR     "GVL"
97  #define SERIALIZED_DAG_HDR      "SDG"
98
99  /*
100  * set structures
101  *
102  * set maintains insertion order of elements
103  * implemented as a singly linked list
104  */
105 struct set_element_t {
106     char                hdr[4];
107     void                *data;
108     struct set_element_t *next;
```

```
109 };
110
111 struct set_t {
112     char                 hdr[4];
113     struct set_element_t *head;
114     struct set_element_t *tail;
115     int                  count;
116     int     (*ele_eq_func)(struct set_element_t *, struct set_element_t *);
117     void    (*ele_print_func)(CSOUND *, struct set_element_t *);
118     struct set_element_t **cache;
119 };
120
121 /* function pointer types for set member equality */
122 typedef int (set_element_data_eq)(struct set_element_t *, struct set_element_t *);
123 int csp_set_element_string_eq(struct set_element_t *ele1, struct set_element_t *ele2);
124 int csp_set_element_ptr_eq(struct set_element_t *ele1, struct set_element_t *ele2);
125
126 /* function pointer types for set member printing */
127 typedef void (set_element_data_print)(CSOUND *, struct set_element_t *);
128 void csp_set_element_string_print(CSOUND *csound, struct set_element_t *ele);
129 void csp_set_element_ptr_print(CSOUND *csound, struct set_element_t *ele);
130
131 /* allocating sets with specification of element equality and printing functions */
132 int csp_set_alloc(CSOUND *csound,
133                   struct set_t **set,
134                   set_element_data_eq *ele_eq_func,
135                   set_element_data_print *ele_print_func);
136 int csp_set_dealloc(CSOUND *csound, struct set_t **set);
137 /* shortcut to get a set of strings uses string element equality and
138  * printing functions */
139 int csp_set_alloc_string(CSOUND *csound, struct set_t **set);
140
141 /* functions to manipulate set, return CSOUND_SUCCESS if successful */
142 int csp_set_add(CSOUND *csound,     struct set_t *set, void *data);
143 int csp_set_remove(CSOUND *csound,  struct set_t *set, void *data);
144 /* check element existance returns 1 if data exists */
145 int csp_set_exists(CSOUND *csound,  struct set_t *set, void *data);
146 int csp_set_print(CSOUND *csound, struct set_t *set);
147
148 /* get a count and access members */
149 int inline csp_set_count(CSOUND *csound, struct set_t *set);
150 int inline csp_set_get_num(CSOUND *csound, struct set_t *set, int num, void **data);
151
152 /*
153  * set union and intersection
154  * allocates a new set in result
155  * union/intersect first and second putting into result
156  */
157 int csp_set_union(CSOUND *csound,
158                   struct set_t *first,
159                   struct set_t *second,
160                   struct set_t **result);
161 int csp_set_intersection(CSOUND *csound,
162                          struct set_t *first,
```

```
163                             struct set_t *second,
164                             struct set_t **result);
165
166 /* spinlock */
167 #if   defined(SPINLOCK_BARRIER)
168 struct barrier_spin_t {
169     int thread_count;
170     int arrived;
171     int spinlock;
172     int lock;
173 };
174 #elif defined(SPINLOCK_2_BARRIER)
175 struct barrier_spin_t {
176     int   thread_count;
177     int   arrived;
178     int   spinlock;
179     int   locks[];
180 };
181 #endif
182
183 #if defined(SPINLOCK_BARRIER) || defined(SPINLOCK_2_BARRIER)
184 /* create a barrier which synchronises thread_count threads */
185 void csp_barrier_alloc(CSOUND *csound,
186                        struct barrier_spin_t **barrier,
187                        int thread_count);
188 void csp_barrier_dealloc(CSOUND *csound, struct barrier_spin_t **barrier);
189 /* calling thread either is the last thread and releases others
190  * or blocks waiting for last thread */
191 void csp_barrier_wait(CSOUND *csound, struct barrier_spin_t *barrier);
192 #endif
193
194 /* semaphore */
195 struct semaphore_spin_t {
196     char    hdr[HDR_LEN];
197     int     thread_count;
198     int     max_threads;
199     int     arrived;
200     int     held;
201     int     spinlock;
202     int     count;
203     int     lock;
204     int     *key;
205     int     locks[];
206 };
207
208 /* create a semaphore with a maximum number of threads
209  * initially 1 thread is allowed in
210  */
211 void csp_semaphore_alloc(CSOUND *csound,
212                          struct semaphore_spin_t **sem,
213                          int max_threads);
214 void csp_semaphore_dealloc(CSOUND *csound, struct semaphore_spin_t **sem);
215 /* wait at the semaphore. if the number allowed in is greater than the
216  * number arrived calling thread continues
```

```
217  * otherwise thread blocks until semaphore is grown
218  */
219 void csp_semaphore_wait(CSOUND *csound, struct semaphore_spin_t *sem);
220 /* increase the number of threads allowed in by 1 */
221 void csp_semaphore_grow(CSOUND *csound, struct semaphore_spin_t *sem);
222 /* reduce the number of threads allowed in and the arrive count by 1
223  * call this when calling thread is finished with the semaphore. */
224 void csp_semaphore_release(CSOUND *csound, struct semaphore_spin_t *sem);
225 /* call when all threads are done with the resource the semaphore is protecting.
226  * releases all blocked threads. */
227 void csp_semaphore_release_end(CSOUND *csound, struct semaphore_spin_t *sem);
228 /* print semaphore info */
229 void csp_semaphore_release_print(CSOUND *csound, struct semaphore_spin_t *sem);
230
231 #endif /* end of include guard: __CS_PAR_BASE_H__ */
```

A.1.2 File: cs_par_orc_semantic_analysis.h

```
 1 #ifndef __CSOUND_ORC_SEMANTIC_ANALYSIS_H__
 2 #define __CSOUND_ORC_SEMANTIC_ANALYSIS_H__
 3
 4 /*
 5  * This module maintains a list of instruments that have been parsed
 6  * When parsing an instrument:
 7  *  csp_orc_sa_instr_add
 8  *      called first to setup instrument (when parsed the instrument name/number)
 9  *  csp_orc_sa_global_read_write_add_list
10  *      called to add globals to that instruments dependency lists
11  *  csp_orc_sa_instr_finalize
12  *      called when finished parsing that instrument
13  *
14  *  csp_orc_sa_instr_get_by_name or by_num
15  *      called to fetch an instrument later
16  */
17
18 /* maintain information about insturments defined */
19 struct instr_semantics_t {
20     char                        hdr[HDR_LEN];
21     char                        *name;
22     int32                       insno;
23     struct set_t                *read;
24     struct set_t                *write;
25     struct set_t                *read_write;
26     uint32_t                    weight;
27     struct instr_semantics_t    *next;
28 };
29
30 void csp_orc_sa_cleanup(CSOUND *csound);
31 void csp_orc_sa_print_list(CSOUND *csound);
32
33 /* maintain state about the current instrument we are parsing */
34 /* add a new instrument */
35 void csp_orc_sa_instr_add(CSOUND *csound, char *name);
36 /* finish the current instrument */
37 void csp_orc_sa_instr_finalize(CSOUND *csound);
38
39 /* add the globals read and written to the current instrument
40  * if write and read contain the same global and size of both is 1 then
41  * that global is added to the read-write list of the current instrument */
```

```
42 void csp_orc_sa_global_read_write_add_list(CSOUND *csound,
43                                            struct set_t *write,
44                                            struct set_t *read);
45
46 /* add to the read and write lists of the current instrument */
47 void csp_orc_sa_global_write_add_list(CSOUND *csound, struct set_t *list);
48 void csp_orc_sa_global_read_add_list(CSOUND *csound, struct set_t *list);
49
50 /* find all the globals contained in the sub-tree node */
51 struct set_t *csp_orc_sa_globals_find(CSOUND *csound, TREE *node);
52
53 /* find an instrument from the instruments parsed */
54 struct instr_semantics_t *csp_orc_sa_instr_get_by_name(char *instr_name);
55 struct instr_semantics_t *csp_orc_sa_instr_get_by_num(int16 insno);
56
57 #endif /* end of include guard: __CSOUND_ORC_SEMANTIC_ANALYSIS_H__ */
```

A.1.3 File: cs_par_dispatch.h

```
1 #ifndef __CS_PAR_DISPATCH_H__
2 #define __CS_PAR_DISPATCH_H__
3
4 /*
5  * locks must first be inserted and then the cache built
6  * following this globals can be locked and unlocked with
7  * the appropriate functions
8  */
9 /* add global locks into AST root */
10 TREE *csp_locks_insert(CSOUND * csound, TREE *root);
11 /* build the cache of global locks */
12 void csp_locks_cache_build(CSOUND *csound);
13 /* lock global with index */
14 void inline csp_locks_lock(CSOUND * csound, int global_index);
15 /* unlock global with index */
16 void inline csp_locks_unlock(CSOUND * csound, int global_index);
17
18 /* fetch a weight for opcode name */
19 uint32_t csp_opcode_weight_fetch(CSOUND *csound, char *name);
20 /* set the time for opcode name */
21 void csp_opcode_weight_set(CSOUND *csound, char *name, double play_time);
22 /* print opcode weights */
23 void csp_weights_dump(CSOUND *csound);
24 /* print opcode weights normalised to 1-100 (inclusive) scale */
25 void csp_weights_dump_normalised(CSOUND *csound);
26 /* dump opcode weights normalised to a file with 1-100 (inclusive) scale
27  * also write out the times associated with the weights */
28 void csp_weights_dump_file(CSOUND *csound);
29 /* load opcode weights from a file */
30 void csp_weights_load(CSOUND *csound);
31 /* calculate the weight for each instrument in the AST
32  * put the weight in the instr_semantics_t structure
33  * stored in cs_par_orc_semantic_analysis */
34 void csp_weights_calculate(CSOUND *csound, TREE *root);
35
36 /* load the parallel decision infomation from specified file */
37 void csp_parallel_compute_spec_setup(CSOUND *csound);
38 /* decide based in parallel decision info whether to do this dag in parallel */
39 int inline csp_parallel_compute_should(CSOUND *csound, struct dag_t *dag);
40
41 /* dag2 prototypes */
42 enum dag_node_type_t {
```

```
43      DAG_NODE_INDV,
44      DAG_NODE_LIST,
45      DAG_NODE_DAG
46  };
47
48  struct dag_base_t {
49      char                        hdr[HDR_LEN];
50      enum dag_node_type_t        type;
51  };
52
53  struct dag_t {
54      struct dag_base_t           hdr;
55
56      int                         count;
57      void                        *mutex;
58      int32_t                     spinlock;
59      int32_t                     table_spinlock;
60      int32_t                     consume_spinlock;
61  #ifdef COUNTING_SEMAPHORE
62      struct semaphore_spin_t     *consume_semaphore;
63  #endif
64      struct dag_node_t           **all;
65      struct dag_node_t           *insds_chain_start;
66      struct dag_node_t           **roots_ori;
67      struct dag_node_t           **roots;
68  #ifndef COUNTING_SEMAPHORE
69      uint8_t                     *root_seen_ori;
70  #endif
71      uint8_t                     *root_seen;
72      int                         *remaining_count_ori;
73      int                         *remaining_count;
74      int                         remaining;
75      int                         first_root_ori;
76      int                         first_root;
77      uint8_t                     **table_ori;
78      uint8_t                     **table;
79
80      /* used for deciding whether to run this dag in parallel */
81      int                         max_roots;
82      uint32_t                    weight;
83  };
84
85  struct dag_node_t {
86      struct dag_base_t           hdr;
87
88      union {
89          struct {
90              struct instr_semantics_t    *instr;
91              INSDS                       *insds;
92              struct dag_node_t           *insds_chain_next;
93          };
94          struct {
95              int count;
96              struct dag_node_t **nodes;
97          };
98      };
99  };
100
101 void csp_dag_alloc(CSOUND *csound, struct dag_t **dag);
102 void csp_dag_dealloc(CSOUND *csound, struct dag_t **dag);
103 /* add a node to the dag with instrument info */
104 void csp_dag_add(CSOUND *csound,
```

```
105                 struct dag_t *dag,
106                 struct instr_semantics_t *instr,
107                 INSDS *insds);
108
109 /* once a DAG has been created and had all its instruments added call this
110  * prepares a DAG for use
111  * builds edges, roots, root countdowns, finds weight, etc */
112 void csp_dag_build(CSOUND *csound, struct dag_t **dag, INSDS *chain);
113 void csp_dag_print(CSOUND *csound, struct dag_t *dag);
114
115 /* return 1 if the DAG is completely consume */
116 int inline csp_dag_is_finished(CSOUND *csound, struct dag_t *dag);
117 /* get a node from the dag
118  * update_hdl should be passed into consume_update when the node has been
119  * performed */
120 void csp_dag_consume(CSOUND *csound,
121                      struct dag_t *dag,
122                      struct dag_node_t **node,
123                      int *update_hdl);
124 /* update the dag having consumed a node previously */
125 void csp_dag_consume_update(CSOUND *csound, struct dag_t *dag, int update_hdl);
126
127 /* get a dag from the cache
128  * if it exists it is retuned
129  * if not builds a new one and stores in the cache, then returns */
130 void csp_dag_cache_fetch(CSOUND *csound, struct dag_t **dag, INSDS *chain);
131 void csp_dag_cache_print(CSOUND *csound);
132
133 #endif /* end of include guard: __CS_PAR_DISPATCH_H__ */
```

A.2 Source Files

A.2.1 File: cs_par_base.c

```
1  /*
2      cs_par_base.c:
3
4      Copyright (C) 2006
5
6      This file is part of Csound.
7
8      The Csound Library is free software; you can redistribute it
9      and/or modify it under the terms of the GNU Lesser General Public
10     License as published by the Free Software Foundation; either
11     version 2.1 of the License, or (at your option) any later version.
12
13     Csound is distributed in the hope that it will be useful,
14     but WITHOUT ANY WARRANTY; without even the implied warranty of
15     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16     GNU Lesser General Public License for more details.
17
18     You should have received a copy of the GNU Lesser General Public
19     License along with Csound; if not, write to the Free Software
20     Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
21     02111-1307 USA
22 */
23
24 #include <stdio.h>
25 #include <stdlib.h>
26 /*#include <string.h>
```

```
27 #include <errno.h>*/
28
29 #include "csoundCore.h"
30 /*#include "namedins.h"*/
31 /* #include "csound_orc.h" */
32 /* #include "tok.h" */
33
34 #include "cs_par_base.h"
35
36 int csp_thread_index_get(CSOUND *csound)
37 {
38 #ifndef mac_classic
39     void *threadId = csound->GetCurrentThreadID();
40
41     int index = 0;
42     THREADINFO *current = csound->multiThreadedThreadInfo;
43
44     if(current == NULL) {
45         return -1;
46     }
47
48     while(current != NULL) {
49         if (pthread_equal(*(pthread_t *)threadId,
50                 *(pthread_t *)current->threadId)) {
51             free(threadId);
52             return index;
53         }
54         index++;
55         current = current->next;
56     }
57 #endif
58     return -1;
59 }
60
61 /**************************************************************************
62  * parallel primitives
63  */
64 #pragma mark -
65 #pragma mark barrier_spin
66
67 #ifdef SPINLOCK_BARRIER
68 int csp_barrier_alloc(CSOUND *csound, struct barrier_spin_t **barrier,
69                       int thread_count)
70 {
71     if (barrier == NULL)
72         csound->Die(csound, "Invalid NULL Parameter barrier");
73     if (thread_count < 1)
74     csound->Die(csound, "Invalid Parameter thread_count must be > 0");
75
76     *barrier = csound->Malloc(csound, sizeof(struct barrier_spin_t));
77     if (*barrier == NULL) {
78         csound->Die(csound, "Failed to allocate barrier");
79     }
80     memset(*barrier, 0, sizeof(struct barrier_spin_t));
81
82     (*barrier)->thread_count = thread_count;
83     /* csoundSpinLock(&((*barrier)->lock)); */
84 }
85
86 int csp_barrier_dealloc(CSOUND *csound, struct barrier_spin_t **barrier)
87 {
88     if (barrier == NULL)
```

```
89          csound->Die(csound, "Invalid NULL Parameter barrier");
90      if (*barrier == NULL)
91          csound->Die(csound, "Invalid NULL Parameter barrier");
92
93      csoundSpinUnLock(&((*barrier)->spinlock));
94      csoundSpinUnLock(&((*barrier)->lock));
95
96      csound->Free(csound, *barrier);
97      *barrier = NULL;
98  }
99
100 int csp_barrier_wait(CSOUND *csound, struct barrier_spin_t *barrier)
101 {
102     /* if (barrier == NULL)
103         csound->Die(csound, "Invalid NULL Parameter barrier"); */
104
105     /* csound->Message(csound, "Barrier Wait Enter\n"); */
106
107     csoundSpinLock(&(barrier->spinlock));
108     barrier->arrived = barrier->arrived + 1;
109     if (barrier->arrived == 1) {
110         csoundSpinLock(&(barrier->lock));
111         csoundSpinUnLock(&(barrier->spinlock));
112         TRACE_1("Blocking First\n");
113         csoundSpinLock(&(barrier->lock));
114         TRACE_1("UnBlocking First\n");
115         csoundSpinUnLock(&(barrier->lock));
116     } else if (barrier->arrived >= barrier->thread_count) {
117         barrier->arrived = 0;
118         TRACE_1("UnBlocking All\n");
119         csoundSpinUnLock(&(barrier->lock));
120         csoundSpinUnLock(&(barrier->spinlock));
121     } else {
122         csoundSpinUnLock(&(barrier->spinlock));
123         TRACE_1("Blocking\n");
124         csoundSpinLock(&(barrier->lock));
125         TRACE_1("UnBlocking\n");
126         csoundSpinUnLock(&(barrier->lock));
127     }
128
129     /* csound->Message(csound, "Barrier Wait Exit\n"); */
130 }
131 #endif
132
133 #ifdef SPINLOCK_2_BARRIER
134 void csp_barrier_alloc(CSOUND *csound, struct barrier_spin_t **barrier,
135                        int thread_count)
136 {
137     if (barrier == NULL)
138         csound->Die(csound, "Invalid NULL Parameter barrier");
139     if (thread_count < 1)
140         csound->Die(csound, "Invalid Parameter thread_count must be > 0");
141
142     *barrier = csound->Malloc(csound, sizeof(struct barrier_spin_t) +
143                                       sizeof(int) * (thread_count - 1));
144     if (*barrier == NULL) {
145         csound->Die(csound, "Failed to allocate barrier");
146     }
147     memset(*barrier, 0, sizeof(struct barrier_spin_t) +
148                             sizeof(int) * (thread_count - 1));
149
150     (*barrier)->thread_count = thread_count;
```

```
151
152     int ctr = 0;
153     while (ctr < (*barrier)->thread_count - 1) {
154         csoundSpinLock(&((*barrier)->locks[ctr]));
155         ctr++;
156     }
157 }
158
159 void csp_barrier_dealloc(CSOUND *csound, struct barrier_spin_t **barrier)
160 {
161     if (barrier == NULL)
162         csound->Die(csound, "Invalid NULL Parameter barrier");
163     if (*barrier == NULL)
164         csound->Die(csound, "Invalid NULL Parameter barrier");
165
166     csoundSpinUnLock(&((*barrier)->spinlock));
167
168     int ctr = 0;
169     while (ctr < (*barrier)->thread_count - 1) {
170         csoundSpinUnLock(&((*barrier)->locks[ctr]));
171         ctr++;
172     }
173     csound->Free(csound, *barrier);
174
175     *barrier = NULL;
176 }
177
178 void csp_barrier_wait(CSOUND *csound, struct barrier_spin_t *barrier)
179 {
180     csoundSpinLock(&(barrier->spinlock));
181     barrier->arrived = barrier->arrived + 1;
182     TRACE_1("BARRIER WAIT n:%i\n", barrier->arrived);
183     if (barrier->arrived >= barrier->thread_count) {
184         TRACE_1("UnBlock All\n");
185         barrier->arrived = 0;
186         int ctr = 0;
187         while (ctr < barrier->thread_count - 1) {
188             TRACE_1("BARRIER UNBLOCKING n:%i\n", ctr);
189             csoundSpinUnLock(&(barrier->locks[ctr]));
190             ctr++;
191         }
192         csoundSpinUnLock(&(barrier->spinlock));
193     } else {
194         int num = barrier->arrived;
195         csoundSpinUnLock(&(barrier->spinlock));
196         TRACE_1("Block\n");
197         csoundSpinLock(&(barrier->locks[num-1]));
198         TRACE_1("UnBlock\n");
199     }
200 }
201 #endif
202
203 /**************************************************************************
204  * semaphore_spin data structure
205  */
206 #pragma mark -
207 #pragma mark semaphore_spin
208
209 void csp_semaphore_alloc(CSOUND *csound, struct semaphore_spin_t **sem,
210                          int max_threads)
211 {
212     if (sem == NULL) csound->Die(csound, "Invalid NULL Parameter sem");
```

```
213
214     *sem = csound->Malloc(csound, sizeof(struct semaphore_spin_t) +
215                                     max_threads * sizeof(int));
216     if (*sem == NULL) {
217         csound->Die(csound, "Failed to allocate barrier");
218     }
219     memset(*sem, 0, sizeof(struct semaphore_spin_t) +
220                         max_threads * sizeof(int));
221     strncpy((*sem)->hdr, SEMAPHORE_HDR, HDR_LEN);
222
223     (*sem)->max_threads = max_threads;
224     (*sem)->key = csound->Malloc(csound, max_threads * sizeof(int));
225     memset((*sem)->key, 0, max_threads * sizeof(int));
226
227     /* int ctr = 0;
228     while (ctr < max_threads) {
229         csoundSpinLock(&((*sem)->locks[ctr]));
230     } */
231 }
232
233 void csp_semaphore_dealloc(CSOUND *csound, struct semaphore_spin_t **sem)
234 {
235     if (sem == NULL) csound->Die(csound, "Invalid NULL Parameter sem");
236     if (*sem == NULL) csound->Die(csound, "Invalid NULL Parameter sem");
237
238     csound->Free(csound, *sem);
239     *sem = NULL;
240 }
241
242 void csp_semaphore_wait(CSOUND *csound, struct semaphore_spin_t *sem)
243 {
244     if (UNLIKELY(sem == NULL))
245         csound->Die(csound, "Invalid NULL Parameter sem");
246
247     csoundSpinLock(&(sem->spinlock));
248
249     /*
250     TRACE_2("[%i] wait\n  arrived: %i\n  threads: %i\n  held:    %i\n  [0]:
251            %i\n  [1]:     %i\n", csp_thread_index_get(csound), sem->arrived,
252            sem->thread_count, sem->held, sem->locks[0], sem->locks[1]);
253     */
254
255     sem->arrived++;
256
257     if (sem->arrived > sem->thread_count) {
258         sem->held++;
259
260         int ctr = 0;
261         while (ctr < sem->max_threads) {
262             if (sem->key[ctr] == 0) {
263                 break;
264             }
265             ctr++;
266         }
267         if (UNLIKELY(ctr >= sem->max_threads)) {
268             csound->Die(csound,
269                         "Should have found a lock to lock. None found.");
270         }
271         sem->key[ctr] = 3;
272         /* int hdl = sem->held - 1;
273         sem->key[hdl] = 1; */
274         TRACE_2("[%i] blocking (%i)\n", csp_thread_index_get(csound), ctr);
```

```
275         csoundSpinLock(&(sem->locks[ctr]));
276         csoundSpinUnLock(&(sem->spinlock));
277
278         csoundSpinLock(&(sem->locks[ctr]));
279         csoundSpinLock(&(sem->spinlock));
280         csoundSpinUnLock(&(sem->locks[ctr]));
281         TRACE_2("[%i] unblocking (%i)\n", csp_thread_index_get(csound), ctr);
282
283         sem->key[ctr] = 0;
284         csoundSpinUnLock(&(sem->spinlock));
285     } else {
286         csoundSpinUnLock(&(sem->spinlock));
287     }
288 }
289
290 void csp_semaphore_grow(CSOUND *csound, struct semaphore_spin_t *sem)
291 {
292     if (UNLIKELY(sem == NULL)) csound->Die(csound, "Invalid NULL Parameter sem");
293
294     csoundSpinLock(&(sem->spinlock));
295
296     /*
297     TRACE_2("[%i] grow\n  arrived: %i\n  threads: %i\n  held:    %i\n  [0]:
298             %i\n  [1]:     %i\n", csp_thread_index_get(csound), sem->arrived,
299             sem->thread_count, sem->held, sem->locks[0], sem->locks[1]);
300     */
301
302     /* csoundSpinUnLock(&(sem->locks[sem->thread_count])); */
303     sem->thread_count++;
304
305     if (sem->held > 0) {
306         int ctr = 0;
307         while (ctr < sem->max_threads) {
308             if (sem->key[ctr] & 2) {
309                 break;
310             }
311             ctr++;
312         }
313         if (LIKELY(ctr < sem->max_threads)) {
314             TRACE_2("[%i] free (%i)\n", csp_thread_index_get(csound), ctr);
315             sem->held--;
316             sem->key[ctr] = 1;
317             csoundSpinUnLock(&(sem->locks[ctr]));
318         } else {
319             csound->Die(csound,
320                         "Should have found a lock to unlock. None found.");
321         }
322     }
323
324     /* int ctr = 0;
325     while (ctr < sem->max_threads) {
326         if (sem->locks[ctr] != 0) {
327             csoundSpinUnLock(&(sem->locks[ctr]));
328             break;
329         }
330     } */
331
332     csoundSpinUnLock(&(sem->spinlock));
333 }
334
335 void csp_semaphore_release(CSOUND *csound, struct semaphore_spin_t *sem)
336 {
```

```
337     if (UNLIKELY(sem == NULL)) csound->Die(csound, "Invalid NULL Parameter sem");
338
339     csoundSpinLock(&(sem->spinlock));
340
341     /*
342     TRACE_2("[%i] release\n  arrived: %i\n  threads: %i\n  held:    %i\n  [0]:
343             %i\n  [1]:     %i\n", csp_thread_index_get(csound), sem->arrived,
344             sem->thread_count, sem->held, sem->locks[0], sem->locks[1]);
345     */
346     sem->arrived--;
347     sem->thread_count--;
348
349     csoundSpinUnLock(&(sem->spinlock));
350 }
351
352 void csp_semaphore_release_end(CSOUND *csound, struct semaphore_spin_t *sem)
353 {
354     if (UNLIKELY(sem == NULL)) csound->Die(csound, "Invalid NULL Parameter sem");
355
356     csoundSpinLock(&(sem->spinlock));
357
358     /*
359     TRACE_2("[%i] release_end\n  arrived: %i\n  threads: %i\n  held:    %i\n
360         [0]:     %i\n  [1]:     %i\n", csp_thread_index_get(csound),
361         sem->arrived, sem->thread_count, sem->held, sem->locks[0],
362         sem->locks[1]);
363     */
364
365     /* csoundSpinUnLock(&(sem->lock)); */
366
367     sem->thread_count++;
368
369     if (sem->held > 0) {
370         int ctr = 0;
371         while (ctr < sem->max_threads) {
372             if (sem->key[ctr] & 2) {
373                 break;
374             }
375             ctr++;
376         }
377         if (LIKELY(ctr < sem->max_threads)) {
378             TRACE_2("[%i] free (%i)\n", csp_thread_index_get(csound), ctr);
379             sem->held--;
380             sem->key[ctr] = 1;
381             csoundSpinUnLock(&(sem->locks[ctr]));
382         } else {
383             csound->Die(csound,
384                         "Should have found a lock to unlock. None found.");
385         }
386     }
387
388     /* int ctr = 0;
389     while (ctr < sem->max_threads) {
390         csoundSpinUnLock(&(sem->locks[ctr]));
391     } */
392
393     csoundSpinUnLock(&(sem->spinlock));
394 }
395
396 void csp_semaphore_release_print(CSOUND *csound, struct semaphore_spin_t *sem)
397 {
398     if (sem == NULL) csound->Die(csound, "Invalid NULL Parameter sem");
```

```
399
400     csoundSpinLock(&(sem->spinlock));
401
402     #define SEMAPHORE_SPIN_BUF 4096
403     char buf[SEMAPHORE_SPIN_BUF];
404     char *bufp = buf;
405
406     bufp = bufp + snprintf(bufp, SEMAPHORE_SPIN_BUF - (bufp - buf),
407                            "Semaphore Spin:\n");
408     bufp = bufp + snprintf(bufp, SEMAPHORE_SPIN_BUF - (bufp - buf),
409                            "  arrived: %i\n", sem->arrived);
410     bufp = bufp + snprintf(bufp, SEMAPHORE_SPIN_BUF - (bufp - buf),
411                            "  threads: %i\n", sem->thread_count);
412     bufp = bufp + snprintf(bufp, SEMAPHORE_SPIN_BUF - (bufp - buf),
413                            "  held:    %i\n", sem->held);
414     /* bufp = bufp + snprintf(bufp, SEMAPHORE_SPIN_BUF - (bufp - buf),
415     "  locked:  %i\n", sem->locks[0]); */
416
417     int ctr = 0;
418     while (ctr < sem->max_threads) {
419         bufp = bufp + snprintf(bufp, SEMAPHORE_SPIN_BUF - (bufp - buf),
420                                "  [%i]:     %i\n", ctr, sem->locks[ctr]);
421         ctr++;
422     }
423
424     csound->Message(csound, "%s", buf);
425
426     csoundSpinUnLock(&(sem->spinlock));
427 }
428
429
430
431 /**************************************************************************
432  * set data structure
433  */
434 #pragma mark -
435 #pragma mark Set
436
437 /* static prototypes */
438 static int set_element_delloc(CSOUND *csound,
439                               struct set_element_t **set_element);
440 static int set_element_alloc(CSOUND *csound,
441                              struct set_element_t **set_element, char *data);
442 static int set_is_set(CSOUND *csound, struct set_t *set);
443 static int set_element_is_set_element(CSOUND *csound,
444                                       struct set_element_t *set_element);
445
446 int csp_set_alloc(CSOUND *csound, struct set_t **set,
447                   set_element_data_eq *ele_eq_func,
448                   set_element_data_print *ele_print_func)
449 {
450     if (set == NULL) csound->Die(csound, "Invalid NULL Parameter set");
451
452     *set = csound->Malloc(csound, sizeof(struct set_t));
453     if (*set == NULL) {
454         csound->Die(csound, "Failed to allocate set");
455     }
456     memset(*set, 0, sizeof(struct set_t));
457     strncpy((*set)->hdr, SET_HDR, HDR_LEN);
458     (*set)->ele_eq_func = ele_eq_func;
459     (*set)->ele_print_func = ele_print_func;
460     (*set)->cache = NULL;
```

```
461
462     return CSOUND_SUCCESS;
463 }
464
465 int csp_set_dealloc(CSOUND *csound, struct set_t **set)
466 {
467     if (set == NULL) csound->Die(csound, "Invalid NULL Parameter set");
468     if (*set == NULL) csound->Die(csound, "Invalid NULL Parameter set");
469     if (!set_is_set(csound, *set))
470         csound->Die(csound, "Invalid Parameter set not a set");
471
472     if ((*set)->cache != NULL) csound->Free(csound, (*set)->cache);
473
474     struct set_element_t *ele = (*set)->head, *next = NULL;
475     while (ele != NULL) {
476         next = ele->next;
477         set_element_delloc(csound, &ele);
478     }
479
480     csound->Free(csound, *set);
481     *set = NULL;
482
483     return CSOUND_SUCCESS;
484 }
485
486 static int set_element_alloc(CSOUND *csound,
487                              struct set_element_t **set_element, char *data)
488 {
489     if (set_element == NULL) csound->Die(csound, "Invalid NULL Parameter set");
490     if (data == NULL) csound->Die(csound, "Invalid NULL Parameter data");
491
492     *set_element = csound->Malloc(csound, sizeof(struct set_element_t));
493     if (*set_element == NULL) {
494         // rc = err_report(RC_ALLOC_FAIL, "Failed to allocate hashtable");
495         csound->Die(csound, "Failed to allocate set element");
496     }
497     memset(*set_element, 0, sizeof(struct set_element_t));
498     strncpy((*set_element)->hdr, SET_ELEMENT_HDR, HDR_LEN);
499     (*set_element)->data = data;
500
501     return CSOUND_SUCCESS;
502 }
503
504 static int set_element_delloc(CSOUND *csound, struct set_element_t **set_element)
505 {
506     if (set_element == NULL)
507         csound->Die(csound, "Invalid NULL Parameter set_element");
508     if (*set_element == NULL)
509         csound->Die(csound, "Invalid NULL Parameter set_element");
510
511     csound->Free(csound, *set_element);
512     *set_element = NULL;
513
514     return CSOUND_SUCCESS;
515 }
516
517 static int set_is_set(CSOUND *csound, struct set_t *set)
518 {
519     if (set == NULL) return 0;
520     char buf[4];
521     strncpy(buf, (char *)set, HDR_LEN);
522     buf[3] = 0;
```

```
523     return strcmp(buf, SET_HDR) == 0;
524 }
525
526 static int set_element_is_set_element(CSOUND *csound,
527                                       struct set_element_t *set_element)
528 {
529     if (set_element == NULL) return 0;
530     char buf[4];
531     strncpy(buf, (char *)set_element, HDR_LEN);
532     buf[3] = 0;
533     return strcmp(buf, SET_ELEMENT_HDR) == 0;
534 }
535
536 int csp_set_alloc_string(CSOUND *csound, struct set_t **set)
537 {
538     return csp_set_alloc(csound, set,
539                          csp_set_element_string_eq,
540                          csp_set_element_string_print);
541 }
542
543 int csp_set_element_string_eq(struct set_element_t *ele1,
544                               struct set_element_t *ele2)
545 {
546     return strcmp((char *)ele1->data, (char *)ele2->data) == 0;
547 }
548
549 int csp_set_element_ptr_eq(struct set_element_t *ele1,
550                            struct set_element_t *ele2)
551 {
552     return (ele1->data == ele2->data);
553 }
554
555 void csp_set_element_string_print(CSOUND *csound, struct set_element_t *ele)
556 {
557     csound->Message(csound, "%s", (char *)ele->data);
558 }
559
560 void csp_set_element_ptr_print(CSOUND *csound,struct set_element_t *ele)
561 {
562     csound->Message(csound, "%p", ele->data);
563 }
564
565 static int set_update_cache(CSOUND *csound, struct set_t *set)
566 {
567     if (set->cache != NULL) {
568         csound->Free(csound, set->cache);
569         set->cache = NULL;
570     }
571     if (set->count > 0) {
572         set->cache = csound->Malloc(csound,
573                                     sizeof(struct set_element_t *) * set->count);
574
575         struct set_element_t *ele = set->head;
576         int ctr = 0;
577         while (ele != NULL) {
578             set->cache[ctr] = ele;
579             ctr++;
580             ele = ele->next;
581         }
582     }
583     return CSOUND_SUCCESS;
584 }
```

```
585
586 /*
587  * if out_set_element is not NULL and the element corresponding to data is not found
588  * it will not be changed
589  */
590 static int set_element_get(CSOUND *csound, struct set_t *set, char *data,
591                            struct set_element_t **out_set_element)
592 {
593 #ifdef SET_DEBUG
594     if (set == NULL) csound->Die(csound, "Invalid NULL Parameter set");
595     if (data == NULL) csound->Die(csound, "Invalid NULL Parameter data");
596     if (out_set_element == NULL)
597         csound->Die(csound, "Invalid NULL Parameter out_set_element");
598 #endif
599
600     struct set_element_t *ele = set->head;
601     struct set_element_t data_ele = { SET_ELEMENT_HDR, data, 0 };
602     while (ele != NULL) {
603         if (set->ele_eq_func(ele, &data_ele)) {
604             *out_set_element = ele;
605             break;
606         }
607         ele = ele->next;
608     }
609     return CSOUND_SUCCESS;
610 }
611
612 int csp_set_add(CSOUND *csound, struct set_t *set, void *data)
613 {
614 #ifdef SET_DEBUG
615     if (set == NULL) csound->Die(csound, "Invalid NULL Parameter set");
616     if (data == NULL) csound->Die(csound, "Invalid NULL Parameter data");
617 #endif
618
619     if (csp_set_exists(csound, set, data)) {
620         return CSOUND_SUCCESS;
621     }
622
623     struct set_element_t *ele = NULL;
624     set_element_alloc(csound, &ele, data);
625     if (set->head == NULL) {
626         set->head = ele;
627         set->tail = ele;
628     } else {
629         set->tail->next = ele;
630         set->tail = ele;
631     }
632     set->count++;
633
634     set_update_cache(csound, set);
635
636     return CSOUND_SUCCESS;
637 }
638
639 int csp_set_remove(CSOUND *csound, struct set_t *set, void *data)
640 {
641 #ifdef SET_DEBUG
642     if (set == NULL) csound->Die(csound, "Invalid NULL Parameter set");
643     if (data == NULL) csound->Die(csound, "Invalid NULL Parameter data");
644 #endif
645
646     struct set_element_t *ele = set->head, *prev = NULL;
```

```
647     struct set_element_t data_ele = { SET_ELEMENT_HDR, data, 0 };
648     while (ele != NULL) {
649         if (set->ele_eq_func(ele, &data_ele)) {
650             if (ele == set->head && ele == set->tail) {
651                 set->head = NULL;
652                 set->tail = NULL;
653             } else if (ele == set->head) {
654                 set->head = ele->next;
655             } else {
656                 prev->next = ele->next;
657             }
658             set_element_delloc(csound, &ele);
659             set->count--;
660             break;
661         }
662         prev = ele;
663         ele = ele->next;
664     }
665
666     set_update_cache(csound, set);
667
668     return CSOUND_SUCCESS;
669 }
670
671 int csp_set_exists(CSOUND *csound, struct set_t *set, void *data)
672 {
673 #ifdef SET_DEBUG
674     if (set == NULL) csound->Die(csound, "Invalid NULL Parameter set");
675     if (data == NULL) csound->Die(csound, "Invalid NULL Parameter data");
676 #endif
677
678     struct set_element_t *ele = NULL;
679     set_element_get(csound, set, data, &ele);
680
681     return (ele == NULL ? 0 : 1);
682 }
683
684 int csp_set_print(CSOUND *csound, struct set_t *set)
685 {
686 #ifdef SET_DEBUG
687     if (set == NULL) csound->Die(csound, "Invalid NULL Parameter set");
688     if (!set_is_set(csound, set))
689         csound->Die(csound, "Invalid Parameter set not a set");
690 #endif
691
692     struct set_element_t *ele = set->head;
693
694     csound->Message(csound, "{ ");
695     while (ele != NULL) {
696         set->ele_print_func(csound, ele);
697
698         TRACE_3(" [%p]", ele);
699
700         if (ele->next != NULL) csound->Message(csound, ", ");
701         ele = ele->next;
702     }
703     csound->Message(csound, " }\n");
704
705     return CSOUND_SUCCESS;
706 }
707
708 int inline csp_set_count(CSOUND *csound, struct set_t *set)
```

```
709 {
710 #ifdef SET_DEBUG
711     if (set == NULL) csound->Die(csound, "Invalid NULL Parameter set");
712     if (!set_is_set(csound, set))
713         csound->Die(csound, "Invalid Parameter set not a set");
714 #endif
715
716     return set->count;
717 }
718
719 /* 0 indexed */
720 int inline csp_set_get_num(CSOUND *csound, struct set_t *set, int num,
721                            void **data)
722 {
723 #ifdef SET_DEBUG
724     if (set == NULL) csound->Die(csound, "Invalid NULL Parameter set");
725     if (!set_is_set(csound, set))
726         csound->Die(csound, "Invalid Parameter set not a set");
727     if (num >= set->count)
728         csound->Die(csound, "Invalid Parameter num is out of bounds");
729     if (data == NULL) csound->Die(csound, "Invalid NULL Parameter data");
730 #endif
731
732     *data = set->cache[num]->data;
733     /*
734     if (set->cache != NULL) {
735
736     } else {
737         int ctr = 0;
738         struct set_element_t *ele = set->head;
739         while (ctr < num && ele != NULL) {
740             ctr++;
741             ele = ele->next;
742         }
743         if (ctr == num && ele != NULL) {
744             *data = ele->data;
745         }
746     }
747     */
748     return CSOUND_SUCCESS;
749 }
750
751 int csp_set_union(CSOUND *csound, struct set_t *first, struct set_t *second,
752                   struct set_t **result)
753 {
754 #ifdef SET_DEBUG
755     if (first == NULL) csound->Die(csound, "Invalid NULL Parameter first");
756     if (!set_is_set(csound, first))
757         csound->Die(csound, "Invalid Parameter set not a first");
758     if (second == NULL) csound->Die(csound, "Invalid NULL Parameter second");
759     if (!set_is_set(csound, second))
760         csound->Die(csound, "Invalid Parameter set not a second");
761     if (result == NULL) csound->Die(csound, "Invalid NULL Parameter result");
762     if (first->ele_eq_func != second->ele_eq_func)
763         csound->Die(csound, "Invalid sets for comparison (different equality)");
764 #endif
765
766     csp_set_alloc(csound, result, first->ele_eq_func, first->ele_print_func);
767
768     int ctr = 0;
769     int first_len = csp_set_count(csound, first);
770     int second_len = csp_set_count(csound, second);
```

```
771
772     while (ctr < first_len) {
773         void *data = NULL;
774         csp_set_get_num(csound, first, ctr, &data);
775         csp_set_add(csound, *result, data);
776         ctr++;
777     }
778
779     ctr = 0;
780     while (ctr < second_len) {
781         void *data = NULL;
782         csp_set_get_num(csound, second, ctr, &data);
783         csp_set_add(csound, *result, data);
784         ctr++;
785     }
786
787     return CSOUND_SUCCESS;
788 }
789
790 int csp_set_intersection(CSOUND *csound, struct set_t *first,
791                          struct set_t *second, struct set_t **result)
792 {
793 #ifdef SET_DEBUG
794     if (first == NULL) csound->Die(csound, "Invalid NULL Parameter first");
795     if (!set_is_set(csound, first))
796         csound->Die(csound, "Invalid Parameter set not a first");
797     if (second == NULL) csound->Die(csound, "Invalid NULL Parameter second");
798     if (!set_is_set(csound, second))
799         csound->Die(csound, "Invalid Parameter set not a second");
800     if (result == NULL) csound->Die(csound, "Invalid NULL Parameter result");
801     if (first->ele_eq_func != second->ele_eq_func)
802         csound->Die(csound, "Invalid sets for comparison (different equality)");
803 #endif
804
805     csp_set_alloc(csound, result, first->ele_eq_func, first->ele_print_func);
806
807     int ctr = 0;
808     int first_len = csp_set_count(csound, first);
809
810     while (ctr < first_len) {
811         void *data = NULL;
812         csp_set_get_num(csound, first, ctr, &data);
813         if (csp_set_exists(csound, second, data)) {
814             csp_set_add(csound, *result, data);
815         }
816         ctr++;
817     }
818
819     return CSOUND_SUCCESS;
820 }
```

A.2.2 File: cs_par_orc_semantic_analysis.c

```
1 /*
2     cs_par_orc_semantic_analysis.c:
3
4     Copyright (C) 2006
5
6     This file is part of Csound.
7
8     The Csound Library is free software; you can redistribute it
```

```
 9      and/or modify it under the terms of the GNU Lesser General Public
10      License as published by the Free Software Foundation; either
11      version 2.1 of the License, or (at your option) any later version.
12
13      Csound is distributed in the hope that it will be useful,
14      but WITHOUT ANY WARRANTY; without even the implied warranty of
15      MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16      GNU Lesser General Public License for more details.
17
18      You should have received a copy of the GNU Lesser General Public
19      License along with Csound; if not, write to the Free Software
20      Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
21      02111-1307 USA
22  */
23
24  #include <stdio.h>
25  #include <stdlib.h>
26  /*#include <string.h>
27  #include <errno.h>*/
28
29  #include "csoundCore.h"
30  /*#include "namedins.h"*/
31  #include "csound_orc.h"
32  #include "tok.h"
33
34  #include "cs_par_base.h"
35  #include "cs_par_orc_semantic_analysis.h"
36
37
38  /**************************************************************************
39   * static function prototypes
40   */
41  /* static int csp_thread_index_get(CSOUND *csound); */
42  static struct instr_semantics_t *instr_semantics_alloc(CSOUND *csound, char *name);
43
44  /**************************************************************************
45   * helper functions
46   */
47
48  static struct instr_semantics_t *instr_semantics_alloc(CSOUND *csound, char *name)
49  {
50      struct instr_semantics_t *instr = csound->Malloc(csound,
51                                                       sizeof(struct instr_semantics_t));
52      memset(instr, 0, sizeof(struct instr_semantics_t));
53      strncpy(instr->hdr, INSTR_SEMANTICS_HDR, HDR_LEN);
54      instr->name = name;
55      instr->insno = -1;
56
57      csp_set_alloc_string(csound, &(instr->read_write));
58      csp_set_alloc_string(csound, &(instr->write));
59      csp_set_alloc_string(csound, &(instr->read));
60
61      return instr;
62  }
63
64  #if 0
65  static int csp_thread_index_get(CSOUND *csound) {
66      int index = 0;
67      void *threadId = csound->GetCurrentThreadID();
68      THREADINFO *current = csound->multiThreadedThreadInfo;
69
70      /* if(current == NULL) {
```

```
71          return -2;
72      }
73
74      while(current != NULL) {
75          if (pthread_equal(*(pthread_t *)threadId, *(pthread_t *)current->threadId)) {
76              return index;
77          }
78          index++;
79          current = current->next;
80      }
81      return -1; */
82
83      int equal = pthread_equal(*(pthread_t *)threadId, *(pthread_t *)current->threadId);
84      free(threadId);
85      return equal;
86  }
87
88  /* #define csp_thread_index_get(csound) 1 */
89  #endif
90
91  /*************************************************************************
92   * parse time support
93   */
94
95  static struct instr_semantics_t *curr;
96  static struct instr_semantics_t *root;
97
98  void csp_orc_sa_cleanup(CSOUND *csound)
99  {
100     struct instr_semantics_t *current = root, *h = NULL;
101     while (current != NULL) {
102
103         csp_set_dealloc(csound, &(current->read));
104         csp_set_dealloc(csound, &(current->write));
105         csp_set_dealloc(csound, &(current->read_write));
106
107         h = current;
108         current = current->next;
109         csound->Free(csound, h);
110     }
111
112     curr = NULL;
113     root = NULL;
114 }
115
116 void csp_orc_sa_print_list(CSOUND *csound)
117 {
118     csound->Message(csound, "Semantic Analysis\n");
119     struct instr_semantics_t *current = root;
120     while (current != NULL) {
121         csound->Message(csound, "Instr: %s\n", current->name);
122         csound->Message(csound, "  read\n");
123         csp_set_print(csound, current->read);
124
125         csound->Message(csound, "  write\n");
126         csp_set_print(csound, current->write);
127
128         csound->Message(csound, "  read_write\n");
129         csp_set_print(csound, current->read_write);
130
131         csound->Message(csound, "  weight: %u\n", current->weight);
132
```

```
133         current = current->next;
134     }
135     csound->Message(csound, "Semantic Analysis Ends\n");
136 }
137
138 void csp_orc_sa_global_read_write_add_list(CSOUND *csound,
139                                            struct set_t *write,
140                                            struct set_t *read)
141 {
142     if (curr == NULL) {
143         csound->Message(csound,
144                         "Add global read, write lists without any instruments\n");
145     } else if (write == NULL  || read == NULL) {
146         csound->Die(csound,
147                     "Invalid NULL parameter set to add to global read, write lists\n");
148     } else {
149         struct set_t *new = NULL;
150         csp_set_union(csound, write, read, &new);
151         if (write->count == 1 && read->count == 1 && new->count == 1) {
152             /* this is a read_write list thing */
153             struct set_t *new_read_write = NULL;
154             csp_set_union(csound, curr->read_write, new, &new_read_write);
155             csp_set_dealloc(csound, &curr->read_write);
156             curr->read_write = new_read_write;
157         } else {
158             csp_orc_sa_global_write_add_list(csound, write);
159             csp_orc_sa_global_read_add_list(csound, read);
160         }
161
162         csp_set_dealloc(csound, &new);
163     }
164 }
165
166 void csp_orc_sa_global_write_add_list(CSOUND *csound, struct set_t *set)
167 {
168     if (curr == NULL) {
169         csound->Message(csound, "Add a global write_list without any instruments\n");
170     } else if (set == NULL) {
171         csound->Die(csound,
172                     "Invalid NULL parameter set to add to a global write_list\n");
173     } else {
174         struct set_t *new = NULL;
175         csp_set_union(csound, curr->write, set, &new);
176
177         csp_set_dealloc(csound, &curr->write);
178         csp_set_dealloc(csound, &set);
179
180         curr->write = new;
181     }
182 }
183
184 void csp_orc_sa_global_read_add_list(CSOUND *csound, struct set_t *set)
185 {
186     if (curr == NULL) {
187         csound->Message(csound,
188                         "add a global read_list without any instruments\n");
189     } else if (set == NULL) {
190         csound->Die(csound,
191                     "Invalid NULL parameter set to add to a global read_list\n");
192     } else {
193         struct set_t *new = NULL;
194         csp_set_union(csound, curr->read, set, &new);
```

```
195
196         csp_set_dealloc(csound, &curr->read);
197         csp_set_dealloc(csound, &set);
198
199         curr->read = new;
200     }
201 }
202
203 static int inInstr = 0;
204
205 void csp_orc_sa_instr_add(CSOUND *csound, char *name)
206 {
207     inInstr = 1;
208     if (root == NULL) {
209         root = instr_semantics_alloc(csound, name);
210         curr = root;
211     } else if (curr == NULL) {
212         struct instr_semantics_t *prev = root;
213         curr = prev->next;
214         while (curr != NULL) {
215             prev = curr;
216             curr = curr->next;
217         }
218         prev->next = instr_semantics_alloc(csound, name);
219         curr = prev->next;
220     } else {
221         curr->next = instr_semantics_alloc(csound, name);
222         curr = curr->next;
223     }
224     // curr->insno = named_instr_find(name);
225 }
226
227 void csp_orc_sa_instr_finalize(CSOUND *csound)
228 {
229     curr = NULL;
230     inInstr = 0;
231 }
232
233 struct set_t *csp_orc_sa_globals_find(CSOUND *csound, TREE *node)
234 {
235     if (node == NULL) {
236         struct set_t *set = NULL;
237         csp_set_alloc_string(csound, &set);
238         return set;
239     }
240
241     struct set_t *left  = csp_orc_sa_globals_find(csound, node->left);
242     struct set_t *right = csp_orc_sa_globals_find(csound, node->right);
243
244     struct set_t *current_set = NULL;
245     csp_set_union(csound, left, right, &current_set);
246
247     csp_set_dealloc(csound, &left);
248     csp_set_dealloc(csound, &right);
249
250     switch (node->type) {
251         case T_IDENT_GI:
252         case T_IDENT_GK:
253         case T_IDENT_GF:
254         case T_IDENT_GW:
255         case T_IDENT_GS:
256         case T_IDENT_GA:
```

```
257             csp_set_add(csound, current_set, node->value->lexeme);
258             break;
259         default:
260               /* no globals */
261             break;
262     }
263
264     if (node->next != NULL) {
265         struct set_t *prev_set = current_set;
266         struct set_t *next = csp_orc_sa_globals_find(csound, node->next);
267         csp_set_union(csound, prev_set, next, &current_set);
268
269         csp_set_dealloc(csound, &prev_set);
270         csp_set_dealloc(csound, &next);
271     }
272
273     return current_set;
274 }
275
276 struct instr_semantics_t *csp_orc_sa_instr_get_by_name(char *instr_name)
277 {
278     struct instr_semantics_t *current_instr = root;
279     while (current_instr != NULL) {
280         if (strcmp(current_instr->name, instr_name) == 0) {
281             return current_instr;
282         }
283         current_instr = current_instr->next;
284     }
285     return NULL;
286 }
287
288 struct instr_semantics_t *csp_orc_sa_instr_get_by_num(int16 insno)
289 {
290     struct instr_semantics_t *current_instr = root;
291     while (current_instr != NULL) {
292         if (current_instr->insno != -1 && current_instr->insno == insno) {
293             return current_instr;
294         }
295         current_instr = current_instr->next;
296     }
297
298     #define BUF_LENGTH 8
299     char buf[BUF_LENGTH];
300     snprintf(buf, BUF_LENGTH, "%i", insno);
301
302     current_instr = csp_orc_sa_instr_get_by_name(buf);
303     if (current_instr != NULL) {
304         current_instr->insno = insno;
305     }
306     return current_instr;
307 }
```

A.2.3 File: cs_par_dispatch.c

```
1 /*
2     cs_par_dispatch.c:
3
4     Copyright (C) 2006
5
6     This file is part of Csound.
7
```

```
 8     The Csound Library is free software; you can redistribute it
 9     and/or modify it under the terms of the GNU Lesser General Public
10     License as published by the Free Software Foundation; either
11     version 2.1 of the License, or (at your option) any later version.
12
13     Csound is distributed in the hope that it will be useful,
14     but WITHOUT ANY WARRANTY; without even the implied warranty of
15     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16     GNU Lesser General Public License for more details.
17
18     You should have received a copy of the GNU Lesser General Public
19     License along with Csound; if not, write to the Free Software
20     Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
21     02111-1307 USA
22 */
23
24 #include <stdio.h>
25 #include <stdlib.h>
26 /*#include <string.h>
27 #include <errno.h>*/
28
29 #include "csoundCore.h"
30 /*#include "namedins.h"*/
31 #include "csound_orc.h"
32 #include "tok.h"
33
34 #include "cs_par_base.h"
35 #include "cs_par_orc_semantic_analysis.h"
36 #include "cs_par_dispatch.h"
37
38 /****************************************************************************
39  * external prototypes not in headers
40  */
41 extern ORCTOKEN *lookup_token(CSOUND *csound, char *);
42
43 /****************************************************************************
44  * static function prototypes
45  */
46 static uint32_t inline hash( uint32_t a );
47 static uint32_t hash_chain(INSDS *chain, uint32_t hash_size);
48 static uint32_t hash_string(char *str, uint32_t hash_size);
49
50 /****************************************************************************
51  * helper functions
52  */
53
54 /* Robert Jenkins' 32 bit integer hash function from
55  * http://www.concentric.net/~Ttwang/tech/inthash.htm
56  */
57 static uint32_t inline hash( uint32_t a )
58 {
59    a = (a+0x7ed55d16) + (a<<12);
60    a = (a^0xc761c23c) ^ (a>>19);
61    a = (a+0x165667b1) + (a<<5);
62    a = (a+0xd3a2646c) ^ (a<<9);
63    a = (a+0xfd7046c5) + (a<<3);
64    a = (a^0xb55a4f09) ^ (a>>16);
65    return a;
66 }
67
68 static uint32_t hash_chain(INSDS *chain, uint32_t hash_size)
69 {
```

```
70      uint32_t val = 0;
71      while (chain != NULL) {
72           val = val ^ chain->insno;
73           val = hash( val );
74           chain = chain->nxtact;
75      }
76      return val % hash_size;
77  }
78
79  static uint32_t hash_string(char *str, uint32_t hash_size)
80  {
81      uint32_t ctr = 0;
82      uint32_t val = 0;
83      uint32_t len = strlen(str);
84      while (ctr < len) {
85           val = val ^ str[ctr];
86           val = hash( val );
87           ctr++;
88      }
89      return val % hash_size;
90  }
91
92  /*****************************************************************************
93   * Global Var Lock Inserts
94   */
95  #pragma mark -
96  #pragma mark Global Var Lock Inserts
97
98  /* global variables lock support */
99  struct global_var_lock_t {
100     char                        hdr[HDR_LEN];
101     char                        *name;
102     int                         index;
103     int32_t                     lock;
104     struct global_var_lock_t    *next;
105 };
106
107 static struct global_var_lock_t *global_var_lock_root;
108 static int global_var_lock_count;
109 static struct global_var_lock_t **global_var_lock_cache;
110
111 void inline csp_locks_lock(CSOUND * csound, int global_index)
112 {
113     if (UNLIKELY(global_index >= global_var_lock_count)) {
114         csound->Die(csound,
115                     "Poorly specified global lock index: %i [max: %i]\n",
116                     global_index, global_var_lock_count);
117     }
118     TRACE_2("Locking:   %i [%p %s]\n", global_index,
119             global_var_lock_cache[global_index],
120             global_var_lock_cache[global_index]->name);
121     csoundSpinLock(&(global_var_lock_cache[global_index]->lock));
122 }
123
124 void inline csp_locks_unlock(CSOUND * csound, int global_index)
125 {
126     if (UNLIKELY(global_index >= global_var_lock_count)) {
127         csound->Die(csound,
128                     "Poorly specified global lock index: %i [max: %i]\n",
129                     global_index, global_var_lock_count);
130     }
131     TRACE_2("UnLocking: %i [%p %s]\n", global_index,
```

```
132             global_var_lock_cache[global_index],
133             global_var_lock_cache[global_index]->name);
134     csoundSpinUnLock(&(global_var_lock_cache[global_index]->lock));
135 }
136
137 static struct global_var_lock_t *global_var_lock_alloc(CSOUND *csound,
138                                                        char *name,
139                                                        int index)
140 {
141     if (name == NULL)
142         csound->Die(csound, "Invalid NULL parameter name for a global variable\n");
143
144     struct global_var_lock_t *ret = csound->Malloc(csound,
145                                                    sizeof(struct global_var_lock_t));
146     memset(ret, 0, sizeof(struct global_var_lock_t));
147     strncpy(ret->hdr, GLOBAL_VAR_LOCK_HDR, HDR_LEN);
148     ret->name = name;
149     ret->index = index;
150
151     global_var_lock_count++;
152
153     return ret;
154 }
155
156 static struct global_var_lock_t *global_var_lock_find(CSOUND *csound, char *name)
157 {
158     if (name == NULL)
159         csound->Die(csound, "Invalid NULL parameter name for a global variable\n");
160
161     if (global_var_lock_root == NULL) {
162         global_var_lock_root = global_var_lock_alloc(csound, name, 0);
163         return global_var_lock_root;
164     } else {
165         struct global_var_lock_t *current = global_var_lock_root, *previous = NULL;
166         int ctr = 0;
167         while (current != NULL) {
168             if (strcmp(current->name, name) == 0) {
169                 break;
170             }
171             previous = current;
172             current = current->next;
173             ctr++;
174         }
175         if (current == NULL) {
176             previous->next = global_var_lock_alloc(csound, name, ctr);
177             return previous->next;
178         } else {
179             return current;
180         }
181     }
182 }
183
184 /* static void locks_print(CSOUND *csound)
185 {
186     csound->Message(csound, "Current Global Locks\n");
187     struct global_var_lock_t *current_global = global_var_lock_root;
188     while (current_global != NULL) {
189         csound->Message(csound, "[%i] %s [%p]\n",
190                         current_global->index, current_global->name, current_global);
191         current_global = current_global->next;
192     }
193 } */
```

```
194
195 TREE *csp_locks_insert(CSOUND *csound, TREE *root)
196 {
197     csound->Message(csound, "Inserting Parallelism Constructs into AST\n");
198
199     TREE *anchor = NULL;
200
201     TREE *current = root;
202     TREE *previous = NULL;
203     struct instr_semantics_t *instr = NULL;
204
205     while(current != NULL) {
206         switch(current->type) {
207             case T_INSTR:
208                 instr = csp_orc_sa_instr_get_by_name(current->left->value->lexeme);
209                 if (instr->read_write->count > 0 &&
210                         instr->read->count == 0 &&
211                         instr->write->count == 0) {
212                     current->right = csp_locks_insert(csound, current->right);
213                 }
214                 break;
215
216             case T_UDO:
217             case T_IF:
218                 break;
219
220             default:
221                 if (current->type == S_ASSIGN) {
222                     struct set_t *left = NULL, *right  = NULL;
223                     left = csp_orc_sa_globals_find(csound, current->left);
224                     right = csp_orc_sa_globals_find(csound, current->right);
225
226                     struct set_t *new = NULL;
227                     csp_set_union(csound, left, right, &new);
228                     /* add locks if this is a read-write global variable
229                      * that is same global read and written in this operation */
230                     if (left->count == 1 && right->count == 1 && new->count == 1) {
231                         char *global_var = NULL;
232                         csp_set_get_num(csound, new, 0, (void **)&global_var);
233
234                         struct global_var_lock_t *gvar =
235                                     global_var_lock_find(csound, global_var);
236
237                         /* add_token(csound, "str", A_TYPE); */
238
239                         char buf[8];
240                         snprintf(buf, 8, "%i", gvar->index);
241
242                         ORCTOKEN *lock_tok   = lookup_token(csound, "globallock");
243                         ORCTOKEN *unlock_tok = lookup_token(csound, "globalunlock");
244                         ORCTOKEN *var_tok    = make_int(csound, buf);
245
246                         TREE *lock_leaf = make_leaf(csound, T_OPCODE, lock_tok);
247                         lock_leaf->right = make_leaf(csound, T_INTGR, var_tok);
248                         TREE *unlock_leaf = make_leaf(csound, T_OPCODE, unlock_tok);
249                         unlock_leaf->right = make_leaf(csound, T_INTGR, var_tok);
250
251                         if (previous == NULL) {
252                             TREE *old_current = lock_leaf;
253                             lock_leaf->next = current;
254                             unlock_leaf->next = current->next;
255                             current->next = unlock_leaf;
```

```
256                             current = old_current;
257                         } else {
258                             previous->next = lock_leaf;
259                             lock_leaf->next = current;
260                             unlock_leaf->next = current->next;
261                             current->next = unlock_leaf;
262                         }
263                     }
264
265                     csp_set_dealloc(csound, &new);
266                     csp_set_dealloc(csound, &left);
267                     csp_set_dealloc(csound, &right);
268                 }
269                 break;
270         }
271
272         if(anchor == NULL) {
273             anchor = current;
274         }
275
276         previous = current;
277         current = current->next;
278
279     }
280
281     csound->Message(csound, "[End Inserting Parallelism Constructs into AST]\n");
282
283     return anchor;
284 }
285
286 void csp_locks_cache_build(CSOUND *csound)
287 {
288     if (global_var_lock_count < 1) {
289         return;
290     }
291
292     global_var_lock_cache = csound->Malloc(csound,
293                                            sizeof(struct global_var_lock_t *) *
294                                            global_var_lock_count);
295
296     int ctr = 0;
297     struct global_var_lock_t *glob = global_var_lock_root;
298     while (glob != NULL && ctr < global_var_lock_count) {
299         global_var_lock_cache[ctr] = glob;
300         glob = glob->next;
301         ctr++;
302     }
303
304     /* csound->Message(csound, "Global Locks Cache\n");
305     ctr = 0;
306     while (ctr < global_var_lock_count) {
307         csound->Message(csound, "[%i] %s\n", global_var_lock_cache[ctr]->index,
308                                 global_var_lock_cache[ctr]->name);
309         ctr++;
310     } */
311 }
312
313
314 /**************************************************************************
315  * weighting
316  */
317 #pragma mark -
```

```
318 #pragma mark Instr weightings
319
320 /* static struct instr_weight_info_t *instr_weight_info_alloc(CSOUND *csound)
321 {
322     struct instr_weight_info_t *ret = csound->Malloc(csound,
323                                                      sizeof(struct instr_weight_info_t));
324     memset(ret, 0, sizeof(struct instr_weight_info_t));
325     strncpy(ret->hdr, INSTR_WEIGHT_INFO_HDR, HDR_LEN);
326
327     return ret;
328 } */
329
330 #define WEIGHT_UNKNOWN_NODE     1
331 #define WEIGHT_S_ASSIGN_NODE    1
332 #define WEIGHT_OPCODE_NODE      5
333
334 static void csp_weights_calculate_instr(CSOUND *csound, TREE *root,
335                                         struct instr_semantics_t *instr)
336 {
337     csound->Message(csound, "Calculating Instrument weight from AST\n");
338
339     TREE *current = root;
340     struct instr_semantics_t *nested_instr = NULL;
341
342     while(current != NULL) {
343         switch(current->type) {
344             case T_INSTR:
345                 nested_instr = csp_orc_sa_instr_get_by_name(current->left->value->lexeme);
346                 /* if (nested_instr->weight == NULL) {
347                     nested_instr->weight = instr_weight_info_alloc(csound);
348                 } */
349                 csp_weights_calculate_instr(csound, current->right, nested_instr);
350                 break;
351
352 #ifdef LOOKUP_WEIGHTS
353             case T_OPCODE:
354             case T_OPCODE0:
355                 instr->weight += csp_opcode_weight_fetch(csound, current->value->lexeme);
356                 break;
357 #else
358             case T_OPCODE:
359             case T_OPCODE0:
360                 instr->weight += WEIGHT_OPCODE_NODE;
361                 break;
362             case S_ASSIGN:
363                 instr->weight += WEIGHT_S_ASSIGN_NODE;
364                 break;
365 #endif
366
367             default:
368                 csound->Message(csound,
369                     "WARNING: Unexpected node type in weight calculation walk %i\n",
370                     current->type);
371                 instr->weight += WEIGHT_UNKNOWN_NODE;
372                 break;
373         }
374
375         current = current->next;
376     }
377
378     csound->Message(csound, "[End Calculating Instrument weight from AST]\n");
379 }
```

```
380
381 void csp_weights_calculate(CSOUND *csound, TREE *root)
382 {
383     csound->Message(csound, "Calculating Instrument weights from AST\n");
384
385     TREE *current = root;
386     struct instr_semantics_t *instr = NULL;
387
388     while(current != NULL) {
389         switch(current->type) {
390             case T_INSTR:
391                 instr = csp_orc_sa_instr_get_by_name(current->left->value->lexeme);
392                 /* if (instr->weight == NULL) {
393                     instr->weight = instr_weight_info_alloc(csound);
394                 } */
395                 csp_weights_calculate_instr(csound, current->right, instr);
396                 break;
397
398             default:
399                 break;
400         }
401
402         current = current->next;
403     }
404
405     csound->Message(csound, "[End Calculating Instrument weights from AST]\n");
406 }
407
408 static void csp_orc_sa_opcode_dump_instr(CSOUND *csound, TREE *root)
409 {
410     TREE *current = root;
411
412     while(current != NULL) {
413         switch(current->type) {
414             case T_INSTR:
415                 break;
416
417             case T_OPCODE:
418             case T_OPCODE0:
419                 csound->Message(csound, "OPCODE: %s\n", current->value->lexeme);
420                 break;
421
422             case S_ASSIGN:
423                 break;
424
425             default:
426                 csound->Message(csound,
427                     "WARNING: Unexpected node type in weight calculation walk %i\n",
428                     current->type);
429                 break;
430         }
431
432         current = current->next;
433     }
434 }
435
436 void csp_orc_sa_opcode_dump(CSOUND *csound, TREE *root)
437 {
438     csound->Message(csound, "Opcode List from AST\n");
439
440     TREE *current = root;
441
```

```
442     while(current != NULL) {
443         switch(current->type) {
444             case T_INSTR:
445                 csp_orc_sa_opcode_dump_instr(csound, current->right);
446                 break;
447 
448             default:
449                 break;
450         }
451 
452         current = current->next;
453     }
454 
455     csound->Message(csound, "[End Opcode List from AST]\n");
456 }
457 
458 
459 /**************************************************************************
460  * weights structure
461  */
462 #pragma mark -
463 #pragma mark weights structure
464 
465 struct opcode_weight_cache_entry_t {
466     uint32_t                            hash_val;
467     struct opcode_weight_cache_entry_t  *next;
468 
469     char                                *name;
470     double                              play_time;
471     uint32_t                            weight;
472 };
473 
474 #define OPCODE_WEIGHT_CACHE_SIZE      128
475 
476 static int opcode_weight_cache_ctr;
477 static struct opcode_weight_cache_entry_t *opcode_weight_cache[OPCODE_WEIGHT_CACHE_SIZE];
478 
479 static int opcode_weight_have_cache;
480 
481 static void opcode_weight_entry_add(CSOUND *csound, char *name, uint32_t weight);
482 
483 static int opcode_weight_entry_alloc(CSOUND *csound,
484                                      struct opcode_weight_cache_entry_t **entry,
485                                      char *name,
486                                      uint32_t weight,
487                                      uint32_t hash_val)
488 {
489     if (entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
490     if (name == NULL) csound->Die(csound, "Invalid NULL Parameter name");
491 
492     *entry = csound->Malloc(csound, sizeof(struct opcode_weight_cache_entry_t));
493     if (*entry == NULL) {
494         csound->Die(csound, "Failed to allocate Opcode Weight cache entry");
495     }
496     memset(*entry, 0, sizeof(struct opcode_weight_cache_entry_t));
497 
498     (*entry)->hash_val = hash_val;
499     (*entry)->name     = name;
500     (*entry)->weight   = weight;
501 
502     return CSOUND_SUCCESS;
503 }
```

```
504
505 static int opcode_weight_entry_dealloc(CSOUND *csound,
506                                         struct opcode_weight_cache_entry_t **entry)
507 {
508     if (entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
509     if (*entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
510
511     csound->Free(csound, *entry);
512     *entry = NULL;
513
514     return CSOUND_SUCCESS;
515 }
516
517 uint32_t csp_opcode_weight_fetch(CSOUND *csound, char *name)
518 {
519     if (name == NULL) csound->Die(csound, "Invalid NULL Parameter name");
520
521     if (opcode_weight_have_cache == 0) {
522         return WEIGHT_OPCODE_NODE;
523     }
524
525     uint32_t hash_val = hash_string(name, OPCODE_WEIGHT_CACHE_SIZE);
526     struct opcode_weight_cache_entry_t *curr = opcode_weight_cache[hash_val];
527     while (curr != NULL) {
528         if (strcmp(curr->name, name) == 0) {
529             return curr->weight;
530         }
531         curr = curr->next;
532     }
533     /* no weight for this opcode use default */
534     csound->Message(csound, "WARNING: no weight found for opcode: %s\n", name);
535     return WEIGHT_OPCODE_NODE;
536 }
537
538 void csp_opcode_weight_set(CSOUND *csound, char *name, double play_time)
539 {
540     if (name == NULL) csound->Die(csound, "Invalid NULL Parameter name");
541
542     if (opcode_weight_have_cache == 0) {
543         return;
544     }
545
546     uint32_t hash_val = hash_string(name, OPCODE_WEIGHT_CACHE_SIZE);
547     /* TRACE_0("Adding %s [%u]\n", name, hash_val); */
548     struct opcode_weight_cache_entry_t *curr = opcode_weight_cache[hash_val];
549
550     while (curr != NULL) {
551         /* TRACE_0("Looking at: %s\n", curr->name); */
552         if (strcmp(curr->name, name) == 0) {
553             if (UNLIKELY(curr->play_time == 0)) {
554                 curr->play_time = play_time;
555             } else {
556                 curr->play_time = 0.9 * curr->play_time + 0.1 * play_time;
557             }
558             return;
559         }
560         curr = curr->next;
561     }
562
563     /* TRACE_0("Not Found Adding\n"); */
564
565     /* not here add it and then set the play time */
```

```
566     opcode_weight_entry_add(csound, strdup(name), WEIGHT_OPCODE_NODE);
567     /* TRACE_0("Not Found Added\n"); */
568     csp_opcode_weight_set(csound, name, play_time);
569     /* TRACE_0("Not Found Done\n"); */
570 }
571
572 static void opcode_weight_entry_add(CSOUND *csound, char *name, uint32_t weight)
573 {
574     if (name == NULL) csound->Die(csound, "Invalid NULL Parameter name");
575
576     uint32_t hash_val = hash_string(name, OPCODE_WEIGHT_CACHE_SIZE);
577     /* TRACE_0("entry_add %s [%u]\n", name, hash_val); */
578     struct opcode_weight_cache_entry_t *curr = opcode_weight_cache[hash_val];
579     int found = 0;
580     while (curr != NULL) {
581         if (strcmp(curr->name, name) == 0) {
582             found = 1;
583             break;
584         }
585         curr = curr->next;
586     }
587     if (found == 0) {
588         /* TRACE_0("Allocing %s\n", name); */
589         opcode_weight_entry_alloc(csound, &curr, name, weight, hash_val);
590         opcode_weight_cache_ctr++;
591
592         curr->next = opcode_weight_cache[hash_val];
593         opcode_weight_cache[hash_val] = curr;
594     }
595 }
596
597 void csp_weights_dump(CSOUND *csound)
598 {
599     if (opcode_weight_have_cache == 0) {
600         csound->Message(csound, "No Weights to Dump (Using Defaults)\n");
601         return;
602     }
603
604     csound->Message(csound, "Weights Dump\n");
605     uint32_t bin_ctr = 0;
606     while (bin_ctr < OPCODE_WEIGHT_CACHE_SIZE) {
607         struct opcode_weight_cache_entry_t *entry = opcode_weight_cache[bin_ctr];
608
609         while (entry != NULL) {
610             csound->Message(csound, "%s => %u\n", entry->name, entry->weight);
611             entry = entry->next;
612         }
613
614         bin_ctr++;
615     }
616     csound->Message(csound, "[Weights Dump end]\n");
617 }
618
619 void csp_weights_dump_file(CSOUND *csound)
620 {
621     if (opcode_weight_have_cache == 0) {
622         csound->Message(csound, "No Weights to Dump to file\n");
623         return;
624     }
625
626     char *path = csound->weights;
627     if (path == NULL) {
```

```
628         return;
629     }
630 
631     FILE *f = fopen(path, "w+");
632     if (f == NULL) {
633         csound->Die(csound, "Opcode Weight Spec File not found at: %s", path);
634     }
635 
636     uint32_t bin_ctr = 0;
637     double min = 0, max = 0;
638     while (bin_ctr < OPCODE_WEIGHT_CACHE_SIZE) {
639         struct opcode_weight_cache_entry_t *entry = opcode_weight_cache[bin_ctr];
640 
641         while (entry != NULL) {
642             if (min == 0) {
643                 min = entry->play_time;
644             }
645 
646             if (entry->play_time != 0 && entry->play_time < min) {
647                 min = entry->play_time;
648             } else if (entry->play_time != 0 && entry->play_time > max) {
649                 max = entry->play_time;
650             }
651 
652             entry = entry->next;
653         }
654 
655         bin_ctr++;
656     }
657 
658     double scale = 99 / (max - min);
659 
660     bin_ctr = 0;
661     while (bin_ctr < OPCODE_WEIGHT_CACHE_SIZE) {
662         struct opcode_weight_cache_entry_t *entry = opcode_weight_cache[bin_ctr];
663 
664         while (entry != NULL) {
665             uint32_t weight = floor((entry->play_time - min) * scale) + 1;
666             fprintf(f, "%s, %u, %g\n", entry->name, weight, entry->play_time);
667             entry = entry->next;
668         }
669 
670         bin_ctr++;
671     }
672 
673     fclose(f);
674 }
675 
676 void csp_weights_dump_normalised(CSOUND *csound)
677 {
678     if (opcode_weight_have_cache == 0) {
679         csound->Message(csound, "No Weights to Dump (Using Defaults)\n");
680         return;
681     }
682 
683     csound->Message(csound, "Weights Dump\n");
684     uint32_t bin_ctr = 0;
685     double min = 0, max = 0;
686     while (bin_ctr < OPCODE_WEIGHT_CACHE_SIZE) {
687         struct opcode_weight_cache_entry_t *entry = opcode_weight_cache[bin_ctr];
688 
689         while (entry != NULL) {
```

```
690             if (min == 0) {
691                 min = entry->play_time;
692             }
693
694             if (entry->play_time != 0 && entry->play_time < min) {
695                 min = entry->play_time;
696             } else if (entry->play_time != 0 && entry->play_time > max) {
697                 max = entry->play_time;
698             }
699
700             entry = entry->next;
701         }
702
703         bin_ctr++;
704     }
705
706     csound->Message(csound, "min: %g\n", min);
707     csound->Message(csound, "max: %g\n", max);
708
709     double scale = 99 / (max - min);
710
711     csound->Message(csound, "scale: %g\n", scale);
712
713     bin_ctr = 0;
714     while (bin_ctr < OPCODE_WEIGHT_CACHE_SIZE) {
715         struct opcode_weight_cache_entry_t *entry = opcode_weight_cache[bin_ctr];
716
717         while (entry != NULL) {
718             uint32_t weight = floor((entry->play_time - min) * scale) + 1;
719             csound->Message(csound, "%s => %u, %g\n",
720                             entry->name, weight, entry->play_time);
721             entry = entry->next;
722         }
723
724         bin_ctr++;
725     }
726
727     if (csound->oparms->calculateWeights) {
728         csp_weights_dump_file(csound);
729     }
730
731     csound->Message(csound, "[Weights Dump end]\n");
732 }
733
734 void csp_weights_load(CSOUND *csound)
735 {
736     char *path = csound->weights;
737     if (path == NULL) {
738         opcode_weight_have_cache = 0;
739         return;
740     }
741     opcode_weight_have_cache = 1;
742
743     memset(opcode_weight_cache, 0, sizeof(struct opcode_weight_cache_entry_t *) *
744                                    OPCODE_WEIGHT_CACHE_SIZE);
745
746     FILE *f = fopen(path, "r");
747     if (f == NULL) {
748         csound->Die(csound, "Opcode Weight Spec File not found at: %s", path);
749     }
750
751     char buf[1024];
```

```
752     char *opcode_name = NULL;
753     int weight = 0;
754     int ctr = 0;
755     int col = 0;
756     int c;
757     while ((c = fgetc(f)) != EOF) {
758         if (col == 0 && c == ',') {
759             col = 1;
760             buf[ctr] = '\0';
761             opcode_name = strdup(buf);
762             ctr = -1;
763         } else if (col == 1 && c == '\n') {
764             col = 0;
765             buf[ctr] = '\0';
766             weight = atoi(buf);
767             opcode_weight_entry_add(csound, opcode_name, weight);
768             opcode_name = NULL; weight = 0;
769             ctr = -1;
770         } else if (col == 1 && c == ',') {
771             col = 2;
772             buf[ctr] = '\0';
773             weight = atoi(buf);
774             ctr = -1;
775         } else if (col == 2 && c == '\n') {
776             col = 0;
777             buf[ctr] = '\0';
778             double play_time = atof(buf);
779             opcode_weight_entry_add(csound, opcode_name, weight);
780             csp_opcode_weight_set(csound, opcode_name, play_time);
781             opcode_name = NULL; weight = 0;
782             ctr = -1;
783         } else {
784             buf[ctr] = c;
785         }
786         ctr++;
787 
788         if (ctr > 1024-1) {
789             // do something about buffer overrun
790         }
791     }
792 
793     fclose(f);
794 }
795 
796 
797 
798 /*****************************************************************************
799  * weighting decision
800  */
801 #pragma mark -
802 #pragma mark dag weighting decision
803 
804 /* static struct instr_weight_info_t *weight_info; */
805 
806 struct weight_decision_info_t {
807     uint32_t    weight_min;
808     uint32_t    weight_max;
809     int         roots_avail_min;
810     int         roots_avail_max;
811 };
812 
813 /* {1125, 0, 2, 0}; */
```

```
814 static struct weight_decision_info_t global_weight_info = {0, 0, 0, 0};
815
816 void csp_orc_sa_parallel_compute_spec_read(CSOUND *csound, const char *path)
817 {
818     FILE *f = fopen(path, "r");
819     if (f == NULL) {
820         csound->Die(csound, "Parallel Spec File not found at: %s", path);
821     }
822
823     int rc = 0;
824     rc = fscanf(f, "%u\n", &(global_weight_info.weight_min));
825     if (rc != 0 || rc == EOF) csound->Die(csound,
826                 "Parallel Spec File invalid format expected weight_min parameter");
827     rc = fscanf(f, "%u\n", &(global_weight_info.weight_max));
828     if (rc != 0 || rc == EOF) csound->Die(csound,
829                 "Parallel Spec File invalid format expected weight_max parameter");
830     rc = fscanf(f, "%i\n", &(global_weight_info.roots_avail_min));
831     if (rc != 0 || rc == EOF) csound->Die(csound,
832                 "Parallel Spec File invalid format expected roots_avail_min parameter");
833     rc = fscanf(f, "%i\n", &(global_weight_info.roots_avail_max));
834     if (rc != 0 || rc == EOF) csound->Die(csound,
835                 "Parallel Spec File invalid format expected roots_avail_max parameter");
836
837     /* todo fix this */
838     fclose(f);
839 }
840
841 void csp_parallel_compute_spec_setup(CSOUND *csound)
842 {
843     char *path = "Default";
844
845     if (csound->weight_info != NULL) {
846         path = csound->weight_info;
847         csp_orc_sa_parallel_compute_spec_read(csound, path);
848     }
849
850     csound->Message(csound, "InstrWeightInfo: [%s]\n"
851                             "   weight_min:      %u\n"
852                             "   weight_max:      %u\n"
853                             "   roots_avail_min: %i\n"
854                             "   roots_avail_max: %i\n",
855                             path,
856                             global_weight_info.weight_min,
857                             global_weight_info.weight_max,
858                             global_weight_info.roots_avail_min,
859                             global_weight_info.roots_avail_max);
860 }
861
862 int inline csp_parallel_compute_should(CSOUND *csound, struct dag_t *dag)
863 {
864     return (dag->weight >= global_weight_info.weight_min &&
865             dag->max_roots >= global_weight_info.roots_avail_min);
866     /* return (dag->ratio > 0.15 && dag->ratio < 0.85); */
867     /* return (dag->weight->weight > weight_info->weight); */
868     /* return 1; */
869 }
870
871 /**************************************************************************
872  * dag2 data structure
873  */
874 #pragma mark -
875 #pragma mark Dag2
```

```
876
877 /* prototypes for dag2 */
878 static void dag_node_2_alloc(CSOUND *csound,
879                              struct dag_node_t **dag_node,
880                              struct instr_semantics_t *instr,
881                              INSDS *insds);
882 static void dag_node_2_alloc_list(CSOUND *csound,
883                                   struct dag_node_t **dag_node,
884                                   int count);
885 static void dag_node_2_dealloc(CSOUND *csound, struct dag_node_t **dag_node);
886
887 /* add all the instruments to a DAG found in an insds chain */
888 static struct dag_t *csp_dag_build_initial(CSOUND *csound, INSDS *chain);
889 /* allocate the dynamic structures which depend on the number of instruments in the dag */
890 static void csp_dag_build_prepare(CSOUND *csound, struct dag_t *dag);
891 /* build the edges (fill out the table) */
892 static void csp_dag_build_edges(CSOUND *csound, struct dag_t *dag);
893 /* find the roots in the DAG and setup the root countdowns */
894 static void csp_dag_build_roots(CSOUND *csound, struct dag_t *dag);
895 /* perpare DAG after building */
896 static void csp_dag_prepare_use_first(CSOUND *csound, struct dag_t *dag);
897 /* perpare DAG after getting out of the cache */
898 static void csp_dag_prepare_use(CSOUND *csound, struct dag_t *dag);
899 /* follow insds chain in nodes to update insds pointers */
900 static inline void csp_dag_prepare_use_insds(CSOUND *csound,
901                                              struct dag_t *dag,
902                                              INSDS *chain);
903
904 #define DAG_NO_LINK     0
905 #define DAG_STRONG_LINK 1
906 #define DAG_WEAK_LINK   2
907
908 void csp_dag_alloc(CSOUND *csound, struct dag_t **dag)
909 {
910     if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
911
912     *dag = csound->Malloc(csound, sizeof(struct dag_t));
913     if (*dag == NULL) {
914         csound->Die(csound, "Failed to allocate dag");
915     }
916     memset(*dag, 0, sizeof(struct dag_t));
917     strncpy((*dag)->hdr.hdr, DAG_2_HDR, HDR_LEN);
918     (*dag)->hdr.type = DAG_NODE_DAG;
919     (*dag)->mutex          = csound->Create_Mutex(0);
920     (*dag)->spinlock       = 0;
921     (*dag)->count          = 0;
922     (*dag)->first_root_ori = -1;
923 #ifdef COUNTING_SEMAPHORE
924     csp_semaphore_alloc(csound, &((*dag)->consume_semaphore),
925                                 csound->oparms->numThreads);
926 #endif
927     /* (*dag)->weight = instr_weight_info_alloc(csound); */
928     (*dag)->weight = 0;
929     (*dag)->max_roots = 0;
930 }
931
932 void csp_dag_dealloc(CSOUND *csound, struct dag_t **dag)
933 {
934     if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
935     if (*dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
936
937     int ctr = 0;
```

```
938     while (ctr < (*dag)->count) {
939         dag_node_2_dealloc(csound, &((*dag)->all[ctr]));
940         (*dag)->all[ctr] = NULL;
941         ctr++;
942     }
943
944     if ((*dag)->all                 != NULL)
945         csound->Free(csound, (*dag)->all);
946     if ((*dag)->roots_ori           != NULL)
947         csound->Free(csound, (*dag)->roots_ori);
948     if ((*dag)->roots               != NULL)
949         csound->Free(csound, (*dag)->roots);
950 #ifndef COUNTING_SEMAPHORE
951     if ((*dag)->root_seen_ori       != NULL)
952         csound->Free(csound, (*dag)->root_seen_ori);
953 #endif
954     if ((*dag)->root_seen           != NULL)
955         csound->Free(csound, (*dag)->root_seen);
956     if ((*dag)->table_ori           != NULL)
957         csound->Free(csound, (*dag)->table_ori);
958     if ((*dag)->table               != NULL)
959         csound->Free(csound, (*dag)->table);
960     if ((*dag)->remaining_count_ori != NULL)
961         csound->Free(csound, (*dag)->remaining_count_ori);
962     if ((*dag)->remaining_count     != NULL)
963         csound->Free(csound, (*dag)->remaining_count);
964
965     /* csound->Free(csound, (*dag)->weight); */
966 #ifdef COUNTING_SEMAPHORE
967     csp_semaphore_dealloc(csound, &((*dag)->consume_semaphore));
968 #endif
969     csound->DestroyMutex((*dag)->mutex);
970
971     csound->Free(csound, *dag);
972     *dag = NULL;
973 }
974
975 static void dag_node_2_alloc(CSOUND *csound,
976                              struct dag_node_t **dag_node,
977                              struct instr_semantics_t *instr,
978                              INSDS *insds)
979 {
980     if (dag_node == NULL) csound->Die(csound, "Invalid NULL Parameter dag_node");
981     if (instr == NULL) csound->Die(csound, "Invalid NULL Parameter instr");
982     if (insds == NULL) csound->Die(csound, "Invalid NULL Parameter insds");
983
984     *dag_node = csound->Malloc(csound, sizeof(struct dag_node_t));
985     if (*dag_node == NULL) {
986         csound->Die(csound, "Failed to allocate dag_node");
987     }
988     memset(*dag_node, 0, sizeof(struct dag_node_t));
989     strncpy((*dag_node)->hdr.hdr, DAG_NODE_2_HDR, HDR_LEN);
990     (*dag_node)->hdr.type = DAG_NODE_INDV;
991     (*dag_node)->instr = instr;
992     (*dag_node)->insds = insds;
993 }
994
995 static void dag_node_2_alloc_list(CSOUND *csound,
996                                   struct dag_node_t **dag_node,
997                                   int count)
998 {
999     if (dag_node == NULL) csound->Die(csound, "Invalid NULL Parameter dag_node");
```

```
1000      if (count <= 0)
1001          csound->Die(csound, "Invalid Parameter count must be greater than 0");
1002
1003      *dag_node = csound->Malloc(csound, sizeof(struct dag_node_t));
1004      if (*dag_node == NULL) {
1005          csound->Die(csound, "Failed to allocate dag_node");
1006      }
1007      memset(*dag_node, 0, sizeof(struct dag_node_t));
1008      strncpy((*dag_node)->hdr.hdr, DAG_NODE_2_HDR, HDR_LEN);
1009      (*dag_node)->hdr.type = DAG_NODE_LIST;
1010      (*dag_node)->nodes = csound->Malloc(csound, sizeof(struct dag_node_t *) * count);
1011      (*dag_node)->count = count;
1012  }
1013
1014  static void dag_node_2_dealloc(CSOUND *csound, struct dag_node_t **dag_node)
1015  {
1016      if (dag_node == NULL) csound->Die(csound, "Invalid NULL Parameter dag_node");
1017      if (*dag_node == NULL) csound->Die(csound, "Invalid NULL Parameter dag_node");
1018
1019      if ((*dag_node)->hdr.type == DAG_NODE_LIST) {
1020          csound->Free(csound, (*dag_node)->nodes);
1021      }
1022
1023      csound->Free(csound, *dag_node);
1024      *dag_node = NULL;
1025  }
1026
1027  void csp_dag_add(CSOUND *csound,
1028                   struct dag_t *dag,
1029                   struct instr_semantics_t *instr,
1030                   INSDS *insds)
1031  {
1032      struct dag_node_t *dag_node = NULL;
1033      dag_node_2_alloc(csound, &dag_node, instr, insds);
1034
1035      struct dag_node_t **old = dag->all;
1036      dag->all = csound->Malloc(csound, sizeof(struct dag_node_t *) *
1037                                        (dag->count + 1));
1038      int ctr = 0;
1039      while (ctr < dag->count) {
1040          dag->all[ctr] = old[ctr];
1041          ctr++;
1042      }
1043      dag->all[ctr] = dag_node;
1044      if (old != NULL) csound->Free(csound, old);
1045      dag->count++;
1046
1047      if (dag->count == 1) {
1048          dag->insds_chain_start = dag->all[0];
1049      } else if (dag->count > 1) {
1050          dag->all[dag->count - 2]->insds_chain_next = dag->all[dag->count - 1];
1051      }
1052  }
1053
1054  static void csp_dag_build_prepare(CSOUND *csound, struct dag_t *dag)
1055  {
1056      if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
1057
1058      if (dag->roots_ori           != NULL) csound->Free(csound, dag->roots_ori);
1059      if (dag->roots               != NULL) csound->Free(csound, dag->roots);
1060  #ifndef COUNTING_SEMAPHORE
1061      if (dag->root_seen_ori       != NULL) csound->Free(csound, dag->root_seen_ori);
```

```
1062 #endif
1063     if (dag->root_seen           != NULL) csound->Free(csound, dag->root_seen);
1064     if (dag->remaining_count_ori != NULL)
1065         csound->Free(csound, dag->remaining_count_ori);
1066     if (dag->remaining_count     != NULL)
1067         csound->Free(csound, dag->remaining_count);
1068     if (dag->table_ori           != NULL) csound->Free(csound, dag->table_ori);
1069     if (dag->table               != NULL) csound->Free(csound, dag->table);
1070
1071     if (dag->count == 0) {
1072         dag->roots_ori           = NULL;
1073         dag->roots               = NULL;
1074 #ifndef COUNTING_SEMAPHORE
1075         dag->root_seen_ori       = NULL;
1076 #endif
1077         dag->root_seen           = NULL;
1078         dag->remaining_count_ori = NULL;
1079         dag->remaining_count     = NULL;
1080         dag->table_ori           = NULL;
1081         dag->table               = NULL;
1082         return;
1083     }
1084
1085     dag->roots_ori           = csound->Malloc(csound, sizeof(struct dag_node_t *) *
1086                                                       dag->count);
1087     dag->roots               = csound->Malloc(csound, sizeof(struct dag_node_t *) *
1088                                                       dag->count);
1089 #ifndef COUNTING_SEMAPHORE
1090     dag->root_seen_ori       = csound->Malloc(csound, sizeof(uint8_t) * dag->count);
1091 #endif
1092     dag->root_seen           = csound->Malloc(csound, sizeof(uint8_t) * dag->count);
1093     dag->remaining_count_ori = csound->Malloc(csound, sizeof(int) * dag->count);
1094     dag->remaining_count     = csound->Malloc(csound, sizeof(int) * dag->count);
1095     dag->table_ori           = csound->Malloc(csound,
1096                                       (sizeof(uint8_t *) * dag->count) +
1097                                       (sizeof(uint8_t) * dag->count * dag->count));
1098     dag->table               = csound->Malloc(csound,
1099                                       (sizeof(uint8_t *) * dag->count) +
1100                                       (sizeof(uint8_t) * dag->count * dag->count));
1101
1102     /* set up pointers for 2D array */
1103     int ctr = 0;
1104     while (ctr < dag->count) {
1105         dag->table_ori[ctr] = ((uint8_t *)dag->table_ori) +
1106                                       (sizeof(uint8_t *) * dag->count) +
1107                                       (sizeof(uint8_t) * dag->count * ctr);
1108         dag->table[ctr]     = ((uint8_t *)dag->table)     +
1109                                       (sizeof(uint8_t *) * dag->count) +
1110                                       (sizeof(uint8_t) * dag->count * ctr);
1111         ctr++;
1112     }
1113
1114     memset(dag->roots_ori,           0,          sizeof(struct dag_node_t *) *
1115                                                    dag->count);
1116     memset(dag->roots,               0,          sizeof(struct dag_node_t *) *
1117                                                    dag->count);
1118 #ifndef COUNTING_SEMAPHORE
1119     memset(dag->root_seen_ori,       0,          sizeof(uint8_t) * dag->count);
1120 #endif
1121     memset(dag->root_seen,           0,          sizeof(uint8_t) * dag->count);
1122     memset(dag->remaining_count_ori, 0,          sizeof(int) * dag->count);
1123     memset(dag->remaining_count,     0,          sizeof(int) * dag->count);
```

```
1124     memset(dag->table_ori[0],        DAG_NO_LINK, sizeof(uint8_t) * dag->count *
1125                                                     dag->count);
1126     memset(dag->table[0],            DAG_NO_LINK, sizeof(uint8_t) * dag->count *
1127                                                     dag->count);
1128 }
1129
1130 static struct dag_t *csp_dag_build_initial(CSOUND *csound, INSDS *chain)
1131 {
1132     if (chain == NULL) csound->Die(csound, "Invalid NULL Parameter chain");
1133
1134     struct dag_t *dag = NULL;
1135     csp_dag_alloc(csound, &dag);
1136
1137     while (chain != NULL) {
1138         struct instr_semantics_t *current_instr =
1139                                         csp_orc_sa_instr_get_by_num(chain->insno);
1140         if (current_instr == NULL) csound->Die(csound,
1141             "Failed to find semantic information for instrument '%i'", chain->insno);
1142
1143         csp_dag_add(csound, dag, current_instr, chain);
1144         dag->weight += current_instr->weight;
1145
1146         chain = chain->nxtact;
1147     }
1148
1149     return dag;
1150 }
1151
1152 static void csp_dag_build_edges(CSOUND *csound, struct dag_t *dag)
1153 {
1154     if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
1155
1156     int dag_root_ctr = 0;
1157     while (dag_root_ctr < dag->count) {
1158         int dag_curr_ctr = dag_root_ctr + 1;
1159         while (dag_curr_ctr < dag->count) {
1160
1161             /* csound->Message(csound, "=== %s <> %s ===\n", dag->all[dag_root_ctr]
1162             ->instr->name, dag->all[dag_curr_ctr]->instr->name); */
1163
1164             int depends = DAG_NO_LINK;
1165             struct set_t *write_intersection = NULL;
1166             csp_set_intersection(csound, dag->all[dag_root_ctr]->instr->write,
1167                                  dag->all[dag_curr_ctr]->instr->read,
1168                                  &write_intersection);
1169             if (csp_set_count(csound, write_intersection) != 0) {
1170                 depends |= DAG_STRONG_LINK;
1171             }
1172             csp_set_dealloc(csound, &write_intersection);
1173
1174             /* csound->Message(csound, "write_intersection depends: %i\n", depends);
1175             csp_set_print(csound, dag->all[dag_root_ctr]->instr->write);
1176             csp_set_print(csound, dag->all[dag_curr_ctr]->instr->read); */
1177
1178             struct set_t *read_intersection = NULL;
1179             csp_set_intersection(csound, dag->all[dag_root_ctr]->instr->read,
1180                                  dag->all[dag_curr_ctr]->instr->write,
1181                                  &read_intersection);
1182             if (csp_set_count(csound, read_intersection) != 0) {
1183                 depends |= DAG_STRONG_LINK;
1184             }
1185             csp_set_dealloc(csound, &read_intersection);
```

```
1186
1187            /* csound->Message(csound, "read_intersection depends: %i\n", depends);
1188            csp_set_print(csound, dag->all[dag_root_ctr]->instr->read);
1189            csp_set_print(csound, dag->all[dag_curr_ctr]->instr->write); */
1190
1191            struct set_t *double_write_intersection = NULL;
1192            csp_set_intersection(csound, dag->all[dag_root_ctr]->instr->write,
1193                                 dag->all[dag_curr_ctr]->instr->write,
1194                                 &double_write_intersection);
1195            if (csp_set_count(csound, double_write_intersection) != 0) {
1196                depends |= DAG_STRONG_LINK;
1197            }
1198            csp_set_dealloc(csound, &double_write_intersection);
1199
1200            /* csound->Message(csound, "double_write_intersection depends: %i\n",
1201                                       depends);
1202            csp_set_print(csound, dag->all[dag_root_ctr]->instr->read);
1203            csp_set_print(csound, dag->all[dag_curr_ctr]->instr->write); */
1204
1205            struct set_t *readwrite_write_intersection = NULL;
1206            csp_set_intersection(csound, dag->all[dag_root_ctr]->instr->read_write,
1207                                 dag->all[dag_curr_ctr]->instr->write,
1208                                 &readwrite_write_intersection);
1209            if (csp_set_count(csound, readwrite_write_intersection) != 0) {
1210                depends |= DAG_STRONG_LINK;
1211            }
1212            csp_set_dealloc(csound, &readwrite_write_intersection);
1213
1214            /* csound->Message(csound, "readwrite_write_intersection depends: %i\n",
1215                                       depends);
1216            csp_set_print(csound, dag->all[dag_root_ctr]->instr->read_write);
1217            csp_set_print(csound, dag->all[dag_curr_ctr]->instr->write); */
1218
1219            struct set_t *readwrite_read_intersection = NULL;
1220            csp_set_intersection(csound, dag->all[dag_root_ctr]->instr->read_write,
1221                                 dag->all[dag_curr_ctr]->instr->read,
1222                                 &readwrite_read_intersection);
1223            if (csp_set_count(csound, readwrite_read_intersection) != 0) {
1224                depends |= DAG_STRONG_LINK;
1225            }
1226            csp_set_dealloc(csound, &readwrite_read_intersection);
1227
1228            /* csound->Message(csound, "readwrite_read_intersection depends: %i\n",
1229                                       depends);
1230            csp_set_print(csound, dag->all[dag_root_ctr]->instr->read_write);
1231            csp_set_print(csound, dag->all[dag_curr_ctr]->instr->write); */
1232
1233            struct set_t *read_readwrite_intersection = NULL;
1234            csp_set_intersection(csound, dag->all[dag_root_ctr]->instr->read,
1235                                 dag->all[dag_curr_ctr]->instr->read_write,
1236                                 &read_readwrite_intersection);
1237            if (csp_set_count(csound, read_readwrite_intersection) != 0) {
1238                depends |= DAG_STRONG_LINK;
1239            }
1240            csp_set_dealloc(csound, &read_readwrite_intersection);
1241
1242            /* csound->Message(csound, "read_readwrite_intersection depends: %i\n",
1243                                       depends);
1244            csp_set_print(csound, dag->all[dag_root_ctr]->instr->read);
1245            csp_set_print(csound, dag->all[dag_curr_ctr]->instr->read_write); */
1246
1247            struct set_t *write_readwrite_intersection = NULL;
```

```
1248             csp_set_intersection(csound, dag->all[dag_root_ctr]->instr->write,
1249                                  dag->all[dag_curr_ctr]->instr->read_write,
1250                                  &write_readwrite_intersection);
1251             if (csp_set_count(csound, write_readwrite_intersection) != 0) {
1252                 depends |= DAG_STRONG_LINK;
1253             }
1254             csp_set_dealloc(csound, &write_readwrite_intersection);
1255
1256             /* csound->Message(csound, "write_readwrite_intersection depends: %i\n",
1257                                    depends);
1258             csp_set_print(csound, dag->all[dag_root_ctr]->instr->write);
1259             csp_set_print(csound, dag->all[dag_curr_ctr]->instr->read_write); */
1260
1261             struct set_t *readwrite_readwrite_intersection = NULL;
1262             csp_set_intersection(csound, dag->all[dag_root_ctr]->instr->read_write,
1263                                  dag->all[dag_curr_ctr]->instr->read_write,
1264                                  &readwrite_readwrite_intersection);
1265             if (csp_set_count(csound, readwrite_readwrite_intersection) != 0) {
1266                 depends |= DAG_WEAK_LINK;
1267             }
1268             csp_set_dealloc(csound, &readwrite_readwrite_intersection);
1269
1270             /* csound->Message(csound, "readwrite_readwrite_intersection depends: %i\n",
1271                                    depends);
1272             csp_set_print(csound, dag->all[dag_root_ctr]->instr->read_write);
1273             csp_set_print(csound, dag->all[dag_curr_ctr]->instr->read_write); */
1274
1275             if (depends & DAG_STRONG_LINK) {
1276                 dag->table_ori[dag_root_ctr][dag_curr_ctr] = DAG_STRONG_LINK;
1277             } else if (depends & DAG_WEAK_LINK) {
1278                 dag->table_ori[dag_root_ctr][dag_curr_ctr] = DAG_WEAK_LINK;
1279             }
1280             dag_curr_ctr++;
1281         }
1282         dag_root_ctr++;
1283     }
1284 }
1285
1286 static void csp_dag_build_roots(CSOUND *csound, struct dag_t *dag)
1287 {
1288     if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
1289
1290     int col = 0;
1291     while (col < dag->count) {
1292         int exists_in = 0;
1293         int row = 0;
1294         while (row < dag->count) {
1295             if (dag->table_ori[row][col] == DAG_STRONG_LINK) {
1296                 exists_in = 1;
1297                 dag->remaining_count_ori[col]++;
1298             }
1299             row++;
1300         }
1301         if (exists_in == 0) {
1302             dag->roots_ori[col] = dag->all[col];
1303 #ifndef COUNTING_SEMAPHORE
1304             dag->root_seen_ori[col] = 1;
1305 #endif
1306             if (dag->first_root_ori == -1) {
1307                 dag->first_root_ori = col;
1308             }
1309         }
```

```
1310         col++;
1311     }
1312 }
1313
1314 static void csp_dag_prepare_use_first(CSOUND *csound, struct dag_t *dag)
1315 {
1316     if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
1317
1318     memcpy(dag->table[0], dag->table_ori[0], sizeof(uint8_t) * dag->count * dag->count);
1319 }
1320
1321 static void csp_dag_prepare_use(CSOUND *csound, struct dag_t *dag)
1322 {
1323     if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
1324
1325     memcpy(dag->roots, dag->roots_ori, sizeof(struct dag_node_t *) * dag->count);
1326 #ifdef COUNTING_SEMAPHORE
1327     memset(dag->root_seen, 0, sizeof(uint8_t) * dag->count);
1328 #else
1329     memcpy(dag->root_seen, dag->root_seen_ori, sizeof(uint8_t) * dag->count);
1330 #endif
1331     memcpy(dag->remaining_count, dag->remaining_count_ori, sizeof(int) * dag->count);
1332     dag->remaining   = dag->count;
1333     dag->first_root  = dag->first_root_ori;
1334 #ifdef COUNTING_SEMAPHORE
1335     dag->root_seen[dag->first_root] = 1;
1336 #else
1337     dag->root_seen[dag->first_root] = 2;
1338 #endif
1339 #ifdef COUNTING_SEMAPHORE
1340     dag->consume_semaphore->thread_count = 1;
1341 #endif
1342 }
1343
1344 static inline void csp_dag_prepare_use_insds(CSOUND *csound,
1345                                              struct dag_t *dag,
1346                                              INSDS *chain)
1347 {
1348     if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
1349
1350     TRACE_2("updating insds\n");
1351     struct dag_node_t *node = dag->insds_chain_start;
1352     INSDS *current_insds = chain;
1353     while (node != NULL && current_insds != NULL) {
1354         TRACE_2("  node: %p, insds: %p\n", node, current_insds);
1355         node->insds = current_insds;
1356         current_insds = current_insds->nxtact;
1357         node = node->insds_chain_next;
1358     }
1359 }
1360
1361 static void csp_dag_calculate_max_roots(CSOUND *csound, struct dag_t *dag)
1362 {
1363     if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
1364
1365     struct instr_semantics_t *instr = NULL;
1366     INSDS *insds = NULL;
1367     struct dag_node_t *node;
1368     int update_hdl = -1;
1369
1370     while (!csp_dag_is_finished(csound, dag)) {
1371         int ctr = 0, roots_avail = 0;
```

```
1372          while (ctr < dag->count) {
1373              if (dag->roots[ctr] != NULL) {
1374                  roots_avail++;
1375              }
1376              ctr++;
1377          }
1378          if (roots_avail > dag->max_roots) {
1379              dag->max_roots = roots_avail;
1380          }
1381          csp_dag_consume(csound, dag, &node, &update_hdl);
1382          instr = node->instr;
1383          insds = node->insds;
1384
1385          if (insds != NULL) {
1386              csp_dag_consume_update(csound, dag, update_hdl);
1387          }
1388      }
1389  }
1390
1391  void csp_dag_build(CSOUND *csound, struct dag_t **dag, INSDS *chain)
1392  {
1393      if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
1394      if (chain == NULL) csound->Die(csound, "Invalid NULL Parameter chain");
1395
1396      *dag = csp_dag_build_initial(csound, chain);
1397      csp_dag_build_prepare(csound, *dag);
1398      csp_dag_build_edges(csound, *dag);
1399      csp_dag_build_roots(csound, *dag);
1400      csp_dag_prepare_use_first(csound, *dag);
1401      csp_dag_prepare_use(csound, *dag);
1402
1403      csp_dag_calculate_max_roots(csound, *dag);
1404      csp_dag_prepare_use(csound, *dag);
1405  }
1406
1407  int inline csp_dag_is_finished(CSOUND *csound, struct dag_t *dag)
1408  {
1409      /* if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag"); */
1410      /* csoundSpinLock(&(dag->spinlock));
1411      int res = (dag->remaining <= 0);
1412      csoundSpinUnLock(&(dag->spinlock));
1413      return res;
1414       */
1415      return (dag->remaining <= 0);
1416  }
1417
1418  /*
1419   * consume an instr and update the first root cache
1420   */
1421  void csp_dag_consume(CSOUND *csound,
1422                       struct dag_t *dag,
1423                       struct dag_node_t **node,
1424                       int *update_hdl)
1425  {
1426      if (UNLIKELY(dag == NULL))
1427          csound->Die(csound, "Invalid NULL Parameter dag");
1428      if (UNLIKELY(node == NULL))
1429          csound->Die(csound, "Invalid NULL Parameter node");
1430      if (UNLIKELY(update_hdl == NULL))
1431          csound->Die(csound, "Invalid NULL Parameter update_hdl");
1432
1433      struct dag_node_t *dag_node = NULL;
```

```
1434     int ctr, first_root;
1435
1436
1437     TRACE_2("[%i] Consuming PreLock [%i, %i] +++++\n",
1438         csp_thread_index_get(csound), dag->first_root, dag->consume_spinlock);
1439
1440 #ifdef COUNTING_SEMAPHORE
1441     csp_semaphore_wait(csound, dag->consume_semaphore);
1442 #else
1443     csoundSpinLock(&(dag->consume_spinlock));
1444 #endif
1445
1446     TRACE_2("[%i] Consuming Have Consume_Spinlock [%i, %i]\n",
1447         csp_thread_index_get(csound), dag->first_root, dag->consume_spinlock);
1448
1449     csoundSpinLock(&(dag->spinlock));
1450
1451     TRACE_2("[%i] Consuming Have Spinlock [%i, %i]\n",
1452         csp_thread_index_get(csound), dag->first_root, dag->consume_spinlock);
1453
1454     if (dag->remaining <= 0) {
1455         TRACE_2("[%i] Consuming Nothing [%i, %i]\n",
1456             csp_thread_index_get(csound), dag->first_root, dag->consume_spinlock);
1457
1458 #ifdef COUNTING_SEMAPHORE
1459         csp_semaphore_release(csound, dag->consume_semaphore);
1460         csp_semaphore_release_end(csound, dag->consume_semaphore);
1461         /* csp_semaphore_grow(csound, dag->consume_semaphore); */
1462 #else
1463         csoundSpinUnLock(&(dag->consume_spinlock));
1464 #endif
1465         csoundSpinUnLock(&(dag->spinlock));
1466         *node = NULL;
1467         *update_hdl = -1;
1468         return;
1469     } else if (UNLIKELY(dag->first_root == -1)) {
1470
1471         csp_dag_print(csound, dag);
1472
1473         csound->Die(csound,
1474             "Expected a root to perform. Found none (%i remaining)", dag->remaining);
1475     }
1476
1477     first_root = dag->first_root;
1478
1479     TRACE_1("[%i] Consuming root:%i\n", csp_thread_index_get(csound), first_root);
1480
1481     dag_node = dag->roots[first_root];
1482     dag->roots[first_root] = NULL;
1483     ctr = 0;
1484     dag->remaining--;
1485     dag->first_root = -1;
1486
1487     if (dag->remaining > 0) {
1488         while (ctr < dag->count) {
1489             if (dag->roots[ctr] != NULL) {
1490                 dag->first_root = ctr;
1491
1492 #ifdef COUNTING_SEMAPHORE
1493                 if (dag->root_seen[ctr] == 0) {
1494                     TRACE_3("[%i] Consuming Unlock [%i] -----\n",
1495                             csp_thread_index_get(csound), dag->first_root);
```

```
1496                     dag->root_seen[ctr] = 1;
1497                     csp_semaphore_grow(csound, dag->consume_semaphore);
1498                 }
1499                 break;
1500 #else
1501                 if (dag->root_seen[ctr] == 1) {
1502                     dag->root_seen[ctr] = 2;
1503                     TRACE_3("[%i] Consuming Unlock [%i] -----\n",
1504                             csp_thread_index_get(csound), dag->first_root);
1505                     csoundSpinUnLock(&(dag->consume_spinlock));
1506                 }
1507                 break;
1508 #endif
1509             }
1510             ctr++;
1511         }
1512     } else {
1513 #ifdef COUNTING_SEMAPHORE
1514         csp_semaphore_release_end(csound, dag->consume_semaphore);
1515 #else
1516         csoundSpinUnLock(&(dag->consume_spinlock));
1517 #endif
1518     }
1519
1520     csoundSpinUnLock(&(dag->spinlock));
1521
1522     *node = dag_node;
1523     *update_hdl = first_root;
1524
1525     TRACE_2("[%i] Consuming Leave [%i, %i]\n",
1526         csp_thread_index_get(csound), dag->first_root, dag->consume_spinlock);
1527 }
1528
1529 /*
1530  * update the roots countdown and roots after consumeing a node
1531  */
1532 void csp_dag_consume_update(CSOUND *csound, struct dag_t *dag, int update_hdl)
1533 {
1534     if (UNLIKELY(dag == NULL)) csound->Die(csound, "Invalid NULL Parameter dag");
1535     if (UNLIKELY(update_hdl < 0 || update_hdl >= dag->count))
1536         csound->Die(csound, "Invalid Parameter update_hdl is outside the DAG range");
1537
1538     csoundSpinLock(&(dag->table_spinlock));
1539
1540     int col = 0;
1541
1542     TRACE_2("[%i] Consuming Update [%i, %i]\n", \
1543         csp_thread_index_get(csound), dag->first_root, dag->consume_spinlock);
1544
1545     while (col < dag->count) {
1546         if (dag->table[update_hdl][col] == DAG_STRONG_LINK) {
1547
1548             /* csoundSpinLock(&(dag->table_spinlock)); */
1549
1550             dag->remaining_count[col]--;
1551             TRACE_3("[%i] Consuming Remaining (%i, %i) [%i, %i]\n",
1552                 csp_thread_index_get(csound), col, dag->remaining_count[col],
1553                 dag->first_root, dag->consume_spinlock);
1554
1555 #ifdef COUNTING_SEMAPHORE
1556             if (dag->remaining_count[col] == 0) {
1557                 csoundSpinLock(&(dag->spinlock));
```

```
1558                dag->roots[col] = dag->all[col];
1559                dag->root_seen[col] = 1;
1560                TRACE_3("[%i] Consuming Found Root [%i, %i]\n",
1561                    csp_thread_index_get(csound), dag->first_root, dag->consume_spinlock);
1562
1563                if (dag->first_root == -1) {
1564                    dag->first_root = col;
1565                    TRACE_3("[%i] Consuming First Root Set [%i, %i]\n",
1566                        csp_thread_index_get(csound), dag->first_root, dag->consume_spinlock);
1567                }
1568                csoundSpinUnLock(&(dag->spinlock));
1569
1570                TRACE_3("[%i] Consuming Unlock [%i, %i] -----\n",
1571                    csp_thread_index_get(csound), dag->first_root, dag->consume_spinlock);
1572                csp_semaphore_grow(csound, dag->consume_semaphore);
1573            }
1574 #else
1575            if (dag->remaining_count[col] == 0) {
1576                csoundSpinLock(&(dag->spinlock));
1577                TRACE_3("[%i] Consuming Found Root [%i]\n",
1578                    csp_thread_index_get(csound), dag->first_root);
1579
1580                if (dag->root_seen[col] == 0) {
1581                    dag->root_seen[col] = 1;
1582                    dag->roots[col] = dag->all[col];
1583                    TRACE_3("[%i] Consuming First Root Set [%i]\n",
1584                        csp_thread_index_get(csound), dag->first_root);
1585                }
1586
1587                if (dag->root_seen[col] == 1 && dag->first_root == -1) {
1588                    dag->first_root = col;
1589
1590                    dag->root_seen[col] = 2;
1591                    TRACE_3("[%i] Consuming Unlock [%i] -----\n",
1592                        csp_thread_index_get(csound), dag->first_root);
1593                    csoundSpinUnLock(&(dag->consume_spinlock));
1594                }
1595                csoundSpinUnLock(&(dag->spinlock));
1596            }
1597 #endif
1598            /* csoundSpinUnLock(&(dag->table_spinlock)); */
1599        }
1600        col++;
1601    }
1602
1603 #ifdef COUNTING_SEMAPHORE
1604    csp_semaphore_release(csound, dag->consume_semaphore);
1605 #endif
1606
1607    csoundSpinUnLock(&(dag->table_spinlock));
1608
1609    TRACE_2("[%i] Consuming Update Leave [%i, %i]\n",
1610        csp_thread_index_get(csound), dag->first_root, dag->consume_spinlock);
1611 }
1612
1613 /* return a string so we can print from multiple threads without interlacing */
1614 static char *csp_dag_string(CSOUND *csound, struct dag_t *dag)
1615 {
1616    #define DAG_2_BUF 4096
1617    char buf[DAG_2_BUF];
1618    char *bufp = buf;
1619
```

```
1620     if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
1621
1622     bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "Dag2:\n");
1623     int ctr = 0;
1624     while (ctr < dag->count) {
1625         if (dag->all[ctr]->hdr.type == DAG_NODE_INDV) {
1626             bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf),
1627                     "  %s [%p]\n", dag->all[ctr]->instr->name, dag->all[ctr]);
1628         } else if (dag->all[ctr]->hdr.type == DAG_NODE_LIST) {
1629             bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "  ");
1630             int inner_ctr = 0;
1631             while (inner_ctr < dag->all[ctr]->count) {
1632                 bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf),
1633                                        "%s [%p] ",
1634                                        dag->all[ctr]->nodes[inner_ctr]->instr->name,
1635                                        dag->all[ctr]->nodes[inner_ctr]);
1636                 inner_ctr++;
1637             }
1638             bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "\n");
1639         }
1640         ctr++;
1641     }
1642
1643     bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "roots:\n");
1644     ctr = 0;
1645     while (ctr < dag->count) {
1646         if (dag->roots[ctr] != NULL) {
1647             if (dag->all[ctr]->hdr.type == DAG_NODE_INDV) {
1648                 bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf),
1649                                        "  %s [%p]\n",
1650                                        dag->roots[ctr]->instr->name,
1651                                        dag->roots[ctr]);
1652             } else if (dag->all[ctr]->hdr.type == DAG_NODE_LIST) {
1653                 bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "  ");
1654                 int inner_ctr = 0;
1655                 while (inner_ctr < dag->roots[ctr]->count) {
1656                     bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf),
1657                                            "%s [%p] ",
1658                                            dag->roots[ctr]->nodes[inner_ctr]->instr->name,
1659                                            dag->roots[ctr]->nodes[inner_ctr]);
1660                     inner_ctr++;
1661                 }
1662                 bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "\n");
1663             }
1664         }
1665         ctr++;
1666     }
1667
1668     bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "root_seen:\n ");
1669     ctr = 0;
1670     while (ctr < dag->count) {
1671         bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf),
1672                                " %hhu ", dag->root_seen[ctr]);
1673         ctr++;
1674     }
1675     bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "\n");
1676
1677     bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "remaining:\n");
1678     ctr = 0;
1679     while (ctr < dag->count) {
1680         bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf),
1681                                "  %i\n", dag->remaining_count[ctr]);
```

```
1682            ctr++;
1683        }
1684
1685        bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf),
1686                               "remaining:      %i\n",  dag->remaining);
1687        bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf),
1688                               "first_root:     %i\n", dag->first_root);
1689
1690        bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "table:\n");
1691        ctr = 0;
1692        while (ctr < dag->count) {
1693            int inner_ctr = 0;
1694            while (inner_ctr < dag->count) {
1695                bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf),
1696                                       "%hhi ", dag->table[ctr][inner_ctr]);
1697                inner_ctr++;
1698            }
1699            bufp = bufp + snprintf(bufp, DAG_2_BUF - (bufp - buf), "\n");
1700            ctr++;
1701        }
1702
1703        return strdup(buf);
1704    }
1705
1706    void csp_dag_print(CSOUND *csound, struct dag_t *dag)
1707    {
1708        char *str = csp_dag_string(csound, dag);
1709        if (str != NULL) {
1710            csound->Message(csound, "%s", str);
1711    #ifdef COUNTING_SEMAPHORE
1712            csp_semaphore_release_print(csound, dag->consume_semaphore);
1713    #endif
1714
1715            free(str);
1716        }
1717    #if 0
1718        if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
1719
1720        csound->Message(csound, "Dag2:\n");
1721        int ctr = 0;
1722        while (ctr < dag->count) {
1723            csound->Message(csound, "  %s\n", dag->all[ctr]->instr->name);
1724            ctr++;
1725        }
1726
1727        csound->Message(csound, "roots:\n");
1728        ctr = 0;
1729        while (ctr < dag->count) {
1730            if (dag->roots[ctr] != NULL) {
1731                csound->Message(csound, "  %s\n", dag->roots[ctr]->instr->name);
1732            }
1733            ctr++;
1734        }
1735
1736        csound->Message(csound, "remaining:      %i\n",  dag->remaining);
1737        csound->Message(csound, "first_root:     %i\n", dag->first_root);
1738        csound->Message(csound, "consume_locked: %i\n", dag->consume_locked);
1739
1740        csound->Message(csound, "table:\n");
1741        ctr = 0;
1742        while (ctr < dag->count) {
1743            int inner_ctr = 0;
```

```
1744         while (inner_ctr < dag->count) {
1745             csound->Message(csound, "%hhi ", dag->table[ctr][inner_ctr]);
1746             inner_ctr++;
1747         }
1748         csound->Message(csound, "\n");
1749         ctr++;
1750     }
1751 #endif
1752 }
1753
1754 /**************************************************************************
1755  * dag2 optimization structure
1756  */
1757 #pragma mark -
1758 #pragma mark Dag2 optimization
1759
1760 static uint64_t dag_opt_counter;
1761
1762 /* attempt to optimize the dag
1763  * this is basis of the combined design */
1764 void csp_dag_optimization(CSOUND *csound, struct dag_t *dag)
1765 {
1766     if (UNLIKELY(dag == NULL)) csound->Die(csound, "Invalid NULL Parameter dag");
1767
1768 #if TRACE > 1
1769     TRACE_0("========== Start ==========\n");
1770     csp_dag_print(csound, dag);
1771 #endif
1772
1773     int starting_row = 0;
1774     int threads = csound->oparms->numThreads;
1775     int end_point = dag->count - threads;
1776     int dag_table_original_size = dag->count;
1777
1778     /* loop through from start to the end of all the rows */
1779     while (starting_row < end_point)
1780     {
1781         TRACE_2("Start Loop\n");
1782         /* struct dag_node_t *readwrite_group[dag->count];
1783         memset(readwrite_group, 0, sizeof(struct dag_node_t *) * dag->count); */
1784         int readwrite_group[dag->count];
1785
1786         /* try and find sqaures of target side length
1787          * decrease target until we find one or its smaller than
1788          * the number of threads
1789          *
1790          * the target square will have instruments performable at the same time
1791          * we place the indexes of the instruments in readwrite_group */
1792         int found_block = 0;
1793         int target = dag->count - starting_row;
1794         while (target >= threads && found_block == 0) {
1795             TRACE_2("Start Loop Target target: %i\n", target);
1796             TRACE_2("end_point: %i\n", end_point);
1797             TRACE_2("starting_row: %i\n", starting_row);
1798
1799             int count_matching_cols = 0;
1800             memset(readwrite_group, -1, sizeof(int) * dag->count);
1801
1802             TRACE_2("readwrite_group done\n");
1803
1804             int row = starting_row, col = starting_row;
1805             while (col < dag->count) {
```

```
1806                 row = starting_row;
1807                 while (row < dag->count &&
1808                         (dag->table_ori[row][col] == DAG_WEAK_LINK ||
1809                          dag->table_ori[row][col] == DAG_NO_LINK)) {
1810                     row++;
1811                 }
1812                 TRACE_2("row: %i\n", row - starting_row);
1813                 if ((row - starting_row) >= target) {
1814                     readwrite_group[count_matching_cols] = col;
1815                     count_matching_cols++;
1816                 }
1817                 col++;
1818             }
1819             TRACE_2("count_matching_cols: %i\n", count_matching_cols);
1820             /* we have a target x target sized square somewhere inside */
1821             if (count_matching_cols >= target) {
1822                 found_block = 1;
1823             } else {  /* go around again looking for a smaller target */
1824                 target--;
1825             }
1826         }
1827
1828         TRACE_2("Found Target? %i (%i)\n", target > threads, target);
1829
1830         /* we've found a target square */
1831         if (target > threads) {
1832             dag_opt_counter++;
1833
1834             /* replace starting_row..starting_row+target with a new row */
1835             int ctr = 0;
1836
1837             /* copy the original dependency table */
1838             uint8_t table_copy[dag_table_original_size][dag_table_original_size];
1839             memcpy(table_copy, dag->table_ori[0],
1840                 sizeof(uint8_t) * dag_table_original_size * dag_table_original_size);
1841
1842             TRACE_2("Table Copy\n");
1843 #if TRACE > 1
1844             ctr = 0;
1845             while (ctr < dag->count) {
1846                 int col_ctr = 0;
1847                 while (col_ctr < dag->count) {
1848                     csound->Message(csound, "%i ", table_copy[ctr][col_ctr]);
1849                     col_ctr++;
1850                 }
1851                 csound->Message(csound, "\n");
1852                 ctr++;
1853             }
1854 #endif
1855
1856             /* stream data structure */
1857             int streams[threads][dag->count];
1858             memset(streams, -1, sizeof(int) * threads * dag->count);
1859             int streams_ix[threads];
1860             memset(streams_ix, 0, sizeof(int) * threads);
1861             int readwrite_group_copy[target];
1862             memcpy(readwrite_group_copy, readwrite_group, sizeof(int) * target);
1863
1864             TRACE_2("Copied Everything\n");
1865 #if TRACE > 1
1866             ctr = 0;
1867             while (ctr < target) {
```

```
1868                csound->Message(csound, "%i: %i\n", ctr, readwrite_group_copy[ctr]);
1869                ctr++;
1870            }
1871 #endif
1872
1873            /* fill out the streams data structure which we use as
1874             * instructions to build the new nodes */
1875            ctr = 0;
1876            while (ctr < target) {
1877                int current_thread = 0;
1878                while (ctr < target && current_thread < threads) {
1879                    int inner_ctr = 0, max_ix = 0;
1880                    uint32_t max_weight = 0;
1881                    while (inner_ctr < target) {
1882                        if (readwrite_group_copy[inner_ctr] >= 0 &&
1883                                dag->all[readwrite_group_copy[inner_ctr]]->instr->weight
1884                                > max_weight) {
1885                            max_weight =
1886                                dag->all[readwrite_group_copy[inner_ctr]]->instr->weight;
1887                            max_ix = inner_ctr;
1888                        }
1889                        inner_ctr++;
1890                    }
1891                    streams[current_thread][streams_ix[current_thread]] =
1892                            readwrite_group_copy[max_ix];
1893                    readwrite_group_copy[max_ix] = -1;
1894                    ctr++;
1895                    streams_ix[current_thread]++;
1896                    current_thread++;
1897                }
1898            }
1899
1900            TRACE_2("Done Stream Instructions\n");
1901 #if TRACE > 1
1902            ctr = 0;
1903            while (ctr < threads) {
1904                csound->Message(csound, "Stream %i: [%i]\n  ", ctr, streams_ix[ctr]);
1905                int stream_ix = 0;
1906                while (stream_ix < streams_ix[ctr]) {
1907                    csound->Message(csound, "%i ", streams[ctr][stream_ix]);
1908                    stream_ix++;
1909                }
1910                csound->Message(csound, "\n");
1911                ctr++;
1912            }
1913 #endif
1914
1915            /* allocate the replacement list dag_nodes */
1916            ctr = 0;
1917            struct dag_node_t *new_nodes[threads];
1918            memset(new_nodes, 0, sizeof(struct dag_node_t *) * threads);
1919            while (ctr < threads) {
1920                dag_node_2_alloc_list(csound, &(new_nodes[ctr]), streams_ix[ctr]);
1921                ctr++;
1922            }
1923
1924            TRACE_2("Allocated all nodes\n");
1925
1926            /* copy nodes from dag into array inside dag_node */
1927            ctr = 0;
1928            while (ctr < threads) {
1929                int inner_ctr = 0;
```

```
1930                while (inner_ctr < new_nodes[ctr]->count) {
1931                    new_nodes[ctr]->nodes[inner_ctr] = dag->all[streams[ctr][inner_ctr]];
1932                    dag->all[streams[ctr][inner_ctr]] = NULL;
1933                    inner_ctr++;
1934                }
1935                ctr++;
1936            }
1937
1938            TRACE_2("Copied all old node references into new nodes\n");
1939
1940            int map_old_to_new_locations[dag->count];
1941            memset(map_old_to_new_locations, -1, sizeof(int) * dag->count);
1942
1943            /* shuffle up all the nodes into the gaps left */
1944            ctr = 0;
1945            int streams_remaining = 0;
1946            while (ctr < dag->count) {
1947                /* we only move things if necessary ie we're in a spot where a
1948                 * node was removed for merging before */
1949                if (dag->all[ctr] == NULL) {
1950                    if (streams_remaining < threads) {  /* slot new nodes in first */
1951                        TRACE_2("Is merged node\n");
1952                        dag->all[ctr] = new_nodes[streams_remaining];
1953
1954                        int stream_ctr = 0;
1955                        while (stream_ctr < streams_ix[streams_remaining]) {
1956                            map_old_to_new_locations[streams[streams_remaining][stream_ctr]]
1957                                = ctr;
1958                            stream_ctr++;
1959                        }
1960
1961                        streams_remaining++;
1962                    } else {                    /* slide down the rest of the nodes */
1963                        TRACE_2("No more merged nodes\n");
1964                        int next = ctr + 1;
1965                        while (next < dag->count && dag->all[next] == NULL) {
1966                            next++;
1967                        }
1968                        if (next < dag->count) {
1969                            TRACE_2("Normal Node\n");
1970                            dag->all[ctr] = dag->all[next];
1971                            dag->all[next] = NULL;
1972
1973                            map_old_to_new_locations[next] = ctr;
1974                        } else {
1975                            TRACE_2("Done copying normal nodes\n");
1976                            /* we're done at this point no more nodes to copy */
1977                            break;
1978                        }
1979                    }
1980                } else {
1981                    map_old_to_new_locations[ctr] = ctr;
1982                }
1983                ctr++;
1984            }
1985
1986            TRACE_2("Shuffled up all nodes\n");
1987
1988 #if TRACE > 1
1989            ctr = 0;
1990            while (ctr < dag->count) {
1991                csound->Message(csound, "%i -> %i\n", ctr, map_old_to_new_locations[ctr]);
```

```
1992                ctr++;
1993            }
1994 #endif
1995
1996            /* empty the table of dependencies */
1997            memset(dag->table_ori[0], 0,
1998                    sizeof(uint8_t) * dag_table_original_size * dag_table_original_size);
1999
2000            /* copy across the new dependency info */
2001            int update_col = 0, update_row = 0;
2002            while (update_col < dag->count) {
2003                update_row = 0;
2004                int working_col = map_old_to_new_locations[update_col];
2005                while (update_row < dag->count) {
2006                    int working_row = map_old_to_new_locations[update_row];
2007
2008                    TRACE_2("(%i, %i) -> (%i, %i)\n",
2009                            update_row, update_col, working_row, working_col);
2010                    TRACE_2("%i -> %i\n",
2011                            table_copy[update_row][update_col],
2012                            dag->table_ori[working_row][working_col]);
2013
2014                    switch (table_copy[update_row][update_col]) {
2015                        case DAG_STRONG_LINK:
2016                            dag->table_ori[working_row][working_col] = DAG_STRONG_LINK;
2017                            break;
2018                        case DAG_WEAK_LINK:
2019                            if (dag->table_ori[working_row][working_col] != DAG_STRONG_LINK) {
2020                                dag->table_ori[working_row][working_col] = DAG_WEAK_LINK;
2021                            }
2022                            break;
2023                        case DAG_NO_LINK:
2024                            if (dag->table_ori[working_row][working_col] != DAG_STRONG_LINK &&
2025                                    dag->table_ori[working_row][working_col] != DAG_WEAK_LINK) {
2026                                dag->table_ori[working_row][working_col] = DAG_NO_LINK;
2027                            }
2028                            break;
2029                    }
2030                    TRACE_2("-> %i\n", dag->table_ori[working_row][working_col]);
2031                    update_row++;
2032                }
2033                update_col++;
2034            }
2035
2036            TRACE_2("Completed dependencies merge\n");
2037
2038            /* update the dag->count */
2039            dag->count = dag->count - target + threads;
2040            starting_row = starting_row + threads;
2041            end_point = dag->count - threads;
2042        } else {
2043            /* not enough parallelism after this instrument */
2044            starting_row++;
2045        }
2046    }
2047
2048    /* reset ready to redo roots and similar things */
2049    memcpy(dag->table[0], dag->table_ori[0],
2050            sizeof(uint8_t) * dag_table_original_size * dag_table_original_size);
2051    memset(dag->roots_ori, 0, sizeof(struct dag_node_t *) * dag_table_original_size);
2052    dag->first_root_ori = -1;
2053
```

```
2054 #ifndef COUNTING_SEMAPHORE
2055     memset(dag->root_seen_ori, 0, sizeof(uint8_t) * dag_table_original_size);
2056 #endif
2057     memset(dag->root_seen, 0, sizeof(uint8_t) * dag_table_original_size);
2058     memset(dag->remaining_count_ori, 0, sizeof(int) * dag_table_original_size);
2059     dag->max_roots = 0;
2060
2061     csp_dag_build_roots(csound, dag);
2062     /* csp_dag_prepare_use_first(csound, dag); */
2063     csp_dag_prepare_use(csound, dag);
2064
2065     csp_dag_calculate_max_roots(csound, dag);
2066     csp_dag_prepare_use(csound, dag);
2067
2068
2069 #if TRACE > 1
2070     TRACE_0("========== New ==========\n");
2071     csp_dag_print(csound, dag);
2072 #endif
2073 }
2074
2075 /*****************************************************************************
2076  * dag2 cache structure
2077  */
2078 #pragma mark -
2079 #pragma mark Dag2 Cache
2080
2081 #ifdef LINEAR_CACHE
2082
2083 struct dag_cache_entry_t {
2084     struct dag_t              *dag;
2085     uint32_t                   uses;
2086     uint32_t                   age;
2087     struct dag_cache_entry_t  *next;
2088     int16                      instrs;
2089     int16                      chain[];
2090 };
2091
2092 static int cache_ctr;
2093 static struct dag_cache_entry_t *cache;
2094
2095 static uint32_t update_ctr;
2096
2097 #define DAG_2_CACHE_SIZE      100
2098 #define DAG_2_DECAY_COMP      1
2099 #define DAG_2_MIN_USE_LIMIT   5000
2100 /* aiming for 8 passes of the cache update before a new entry must exist solely on its usage */
2101 #define DAG_2_MIN_AGE_LIMIT   256
2102 #define DAG_2_AGE_START       131072
2103
2104 static int csp_dag_cache_entry_dealloc(CSOUND *csound,
2105                                        struct dag_cache_entry_t **entry);
2106 static int csp_dag_cache_entry_alloc(CSOUND *csound,
2107                                      struct dag_cache_entry_t **entry,
2108                                      INSDS *chain);
2109 static int csp_dag_cache_compare(CSOUND *csound,
2110                                  struct dag_cache_entry_t *entry,
2111                                  INSDS *chain);
2112 static void csp_dag_cache_update(CSOUND *csound);
2113
2114 void csp_dag_cache_print(CSOUND *csound)
2115 {
```

```
2116     csound->Message(csound, "Dag2 Cache Size: %i\n", cache_ctr);
2117     uint32_t sum = 0, max = 0, min = UINT32_MAX, sum_age = 0;
2118     struct dag_cache_entry_t *entry = cache, *prev = NULL;
2119     while (entry != NULL) {
2120         if (entry->uses > max) max = entry->uses;
2121         else if (entry->uses < min) min = entry->uses;
2122         sum = sum + entry->uses;
2123         sum_age = sum_age + entry->age;
2124         entry = entry->next;
2125     }
2126     csound->Message(csound, "Dag2 Avg Uses: %u\n", sum / cache_ctr);
2127     csound->Message(csound, "Dag2 Min Uses: %u\n", min);
2128     csound->Message(csound, "Dag2 Max Uses: %u\n", max);
2129     csound->Message(csound, "Dag2 Avg Age: %u\n", sum_age / cache_ctr);
2130     csound->Message(csound, "Dag2 Fetches:  %u\n", update_ctr);
2131 }
2132
2133 static int csp_dag_cache_entry_alloc(CSOUND *csound,
2134                                      struct dag_cache_entry_t **entry,
2135                                      INSDS *chain)
2136 {
2137     if (entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
2138     if (chain == NULL) csound->Die(csound, "Invalid NULL Parameter chain");
2139
2140     int ctr = 0;
2141     INSDS *current_insds = chain;
2142     while (current_insds != NULL) {
2143         ctr++;
2144         current_insds = current_insds->nxtact;
2145     }
2146
2147     *entry = csound->Malloc(csound, sizeof(struct dag_cache_entry_t) + sizeof(int16) * ctr);
2148     if (*entry == NULL) {
2149         csound->Die(csound, "Failed to allocate Dag2 cache");
2150     }
2151     memset(*entry, 0, sizeof(struct dag_cache_entry_t) + sizeof(int16) * ctr);
2152     (*entry)->uses   = 1;
2153     (*entry)->age    = DAG_2_AGE_START;
2154     (*entry)->instrs = ctr;
2155
2156     ctr = 0;
2157     current_insds = chain;
2158     while (current_insds != NULL) {
2159         (*entry)->chain[ctr] = current_insds->insno;
2160         ctr++;
2161         current_insds = current_insds->nxtact;
2162     }
2163
2164     struct dag_t *dag = NULL;
2165     csp_dag_build(csound, &dag, chain);
2166     (*entry)->dag = dag;
2167
2168     return CSOUND_SUCCESS;
2169 }
2170
2171 static int csp_dag_cache_entry_dealloc(CSOUND *csound, struct dag_cache_entry_t **entry)
2172 {
2173     if (entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
2174     if (*entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
2175
2176     csp_dag_dealloc(csound, &((*entry)->dag));
2177
```

```
2178     csound->Free(csound, *entry);
2179     *entry = NULL;
2180
2181     return CSOUND_SUCCESS;
2182 }
2183
2184 static void csp_dag_cache_update(CSOUND *csound)
2185 {
2186     if (cache_ctr < DAG_2_CACHE_SIZE) {
2187         return;
2188     }
2189     csound->Message(csound, "Cache Update\n");
2190     struct dag_cache_entry_t *entry = cache, *prev = NULL;
2191     while (entry != NULL) {
2192         entry->uses = entry->uses >> DAG_2_DECAY_COMP;
2193         entry->age  = entry->age  >> DAG_2_DECAY_COMP;
2194         if (entry->uses < DAG_2_MIN_USE_LIMIT &&
2195                 entry->age < DAG_2_MIN_AGE_LIMIT && prev != NULL) {
2196             prev->next = entry->next;
2197             csp_dag_cache_entry_dealloc(csound, &entry);
2198             entry = prev->next;
2199             cache_ctr--;
2200         } else {
2201             prev = entry;
2202             entry = entry->next;
2203         }
2204     }
2205 }
2206
2207 static int csp_dag_cache_compare(CSOUND *csound,
2208                                  struct dag_cache_entry_t *entry,
2209                                  INSDS *chain)
2210 {
2211     /* if (entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
2212     if (chain == NULL) csound->Die(csound, "Invalid NULL Parameter chain"); */
2213
2214     INSDS *current_insds = chain;
2215     int32_t ctr = 0;
2216     while (current_insds != NULL && ctr < entry->instrs) {
2217         if (current_insds->insno != entry->chain[ctr]) {
2218             return 0;
2219         }
2220         current_insds = current_insds->nxtact;
2221         ctr++;
2222     }
2223     if (ctr >= entry->instrs && current_insds != NULL) {
2224         return 0;
2225     } else if (ctr < entry->instrs && current_insds == NULL) {
2226         return 0;
2227     } else {
2228         return 1;
2229     }
2230 }
2231
2232 void csp_dag_cache_fetch(CSOUND *csound, struct dag_t **dag, INSDS *chain)
2233 {
2234     if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
2235     if (chain == NULL) csound->Die(csound, "Invalid NULL Parameter chain");
2236
2237     /* static int ctr = 0; */
2238
2239     update_ctr++;
```

```
2240     if (update_ctr == 10000) {
2241         csp_dag_cache_update(csound);
2242         update_ctr = 0;
2243     }
2244
2245     struct dag_cache_entry_t *curr = cache;
2246     while (curr != NULL) {
2247         if (csp_dag_cache_compare(csound, curr, chain)) {
2248             TRACE_2("Cache Hit [%i]\n", cache_ctr);
2249             *dag = curr->dag;
2250
2251             curr->uses++;
2252
2253             csp_dag_prepare_use_insds(csound, curr->dag, chain);
2254             csp_dag_prepare_use(csound, curr->dag);
2255             break;
2256         }
2257         curr = curr->next;
2258     }
2259     if (*dag == NULL) {
2260         csp_dag_cache_entry_alloc(csound, &curr, chain);
2261         cache_ctr++;
2262         *dag = curr->dag;
2263         curr->next = cache;
2264         cache = curr;
2265
2266         TRACE_2("Cache Miss [%i]\n", cache_ctr);
2267     }
2268 }
2269
2270 #endif
2271
2272
2273 #ifdef HASH_CACHE
2274
2275 struct dag_cache_entry_t {
2276     uint32_t                    hash_val;
2277     struct dag_cache_entry_t  *next;
2278     struct dag_t              *dag;
2279     uint32_t                    uses;
2280     uint32_t                    age;
2281
2282     int16                       instrs;
2283     int16                       chain[];
2284 };
2285
2286 #define DAG_2_CACHE_SIZE     128
2287 #define DAG_2_DECAY_COMP     1
2288 #define DAG_2_MIN_USE_LIMIT  5000
2289 /* aiming for 8 passes of the cache update before a
2290  * new entry must exist solely on its usage */
2291 #define DAG_2_MIN_AGE_LIMIT  256
2292 #define DAG_2_AGE_START      131072
2293
2294 static int cache_ctr;
2295 static struct dag_cache_entry_t *cache[DAG_2_CACHE_SIZE];
2296
2297 #ifdef HYBRID_HASH_CACHE
2298 static struct dag_cache_entry_t *cache_last;
2299 #endif
2300
2301 static uint32_t update_ctr;
```

```
2302
2303 static int csp_dag_cache_entry_alloc(CSOUND *csound,
2304                                       struct dag_cache_entry_t **entry,
2305                                       INSDS *chain,
2306                                       uint32_t hash_val);
2307 static int csp_dag_cache_entry_dealloc(CSOUND *csound,
2308                                        struct dag_cache_entry_t **entry);
2309 static int csp_dag_cache_compare(CSOUND *csound,
2310                                  struct dag_cache_entry_t *entry,
2311                                  INSDS *chain);
2312 static void csp_dag_cache_update(CSOUND *csound);
2313 static void csp_dag_cache_print_weights_dump(CSOUND *csound);
2314
2315 void csp_dag_cache_print(CSOUND *csound)
2316 {
2317     csound->Message(csound, "Dag2 Cache Size: %i\n", cache_ctr);
2318     uint32_t sum = 0, max = 0, min = UINT32_MAX, sum_age = 0;
2319     uint32_t weight_sum = 0, weight_max = 0, weight_min = UINT32_MAX;
2320     uint32_t instr_num_sum = 0, instr_num_max = 0, instr_num_min = UINT32_MAX;
2321     uint32_t root_avail_sum = 0, root_avail_max = 0, root_avail_min = UINT32_MAX;
2322     uint32_t bin_ctr = 0, bins_empty = 0, bins_used = 0, bin_max = 0;
2323     while (bin_ctr < DAG_2_CACHE_SIZE) {
2324         struct dag_cache_entry_t *entry = cache[bin_ctr];
2325
2326         if (entry == NULL) bins_empty++;
2327         else bins_used++;
2328
2329         uint32_t entry_ctr = 0;
2330         while (entry != NULL) {
2331             entry_ctr++;
2332             if (entry->uses > max) max = entry->uses;
2333             else if (entry->uses < min) min = entry->uses;
2334             sum = sum + entry->uses;
2335             sum_age = sum_age + entry->age;
2336
2337             weight_sum += entry->dag->weight;
2338             if (entry->dag->weight > weight_max) weight_max = entry->dag->weight;
2339             else if (entry->dag->weight < weight_min) weight_min = entry->dag->weight;
2340
2341             instr_num_sum += entry->instrs;
2342             if (entry->instrs > instr_num_max) instr_num_max = entry->instrs;
2343             else if (entry->instrs < instr_num_min) instr_num_min = entry->instrs;
2344
2345             root_avail_sum += entry->dag->max_roots;
2346             if (entry->dag->max_roots > root_avail_max) root_avail_max = entry->dag->max_roots;
2347             else if (entry->dag->max_roots < root_avail_min)
2348                 root_avail_min = entry->dag->max_roots;
2349
2350             entry = entry->next;
2351         }
2352
2353         if (entry_ctr > bin_max) bin_max = entry_ctr;
2354
2355         bin_ctr++;
2356     }
2357     csound->Message(csound, "Dag2 Avg Uses: %u\n", sum / cache_ctr);
2358     csound->Message(csound, "Dag2 Min Uses: %u\n", min);
2359     csound->Message(csound, "Dag2 Max Uses: %u\n", max);
2360     csound->Message(csound, "Dag2 Avg Age: %u\n", sum_age / cache_ctr);
2361     csound->Message(csound, "Dag2 Fetches:  %u\n", update_ctr);
2362
2363     csound->Message(csound, "Dag2 Empty Bins:  %u\n", bins_empty);
```

```
2364      csound->Message(csound, "Dag2 Used Bins:  %u\n", bins_used);
2365      csound->Message(csound, "Dag2 Bins Max:  %u\n", bin_max);
2366      csound->Message(csound, "Dag2 Bins Avg:  %u\n", cache_ctr / bins_used);
2367
2368      csound->Message(csound, "Weights Avg: %u\n", weight_sum / cache_ctr);
2369      csound->Message(csound, "Weights Min: %u\n", weight_min);
2370      csound->Message(csound, "Weights Max: %u\n", weight_max);
2371      csound->Message(csound, "Weights InstrNum Avg: %u\n", instr_num_sum / cache_ctr);
2372      csound->Message(csound, "Weights InstrNum Min: %u\n", instr_num_min);
2373      csound->Message(csound, "Weights InstrNum Max: %u\n", instr_num_max);
2374      csound->Message(csound, "Roots Available Avg: %u\n", root_avail_sum / cache_ctr);
2375      csound->Message(csound, "Roots Available Min: %u\n", root_avail_min);
2376      csound->Message(csound, "Roots Available Max: %u\n", root_avail_max);
2377
2378      csound->Message(csound, "Number Optimized: %llu\n", dag_opt_counter);
2379
2380
2381      if (csound->weight_dump != NULL) {
2382          csp_dag_cache_print_weights_dump(csound);
2383      }
2384  }
2385
2386  static void csp_dag_cache_print_weights_dump(CSOUND *csound)
2387  {
2388      char *path = csound->weight_dump;
2389
2390      FILE *f = fopen(path, "w+");
2391      if (f == NULL) {
2392          csound->Die(csound, "Parallel Dump File not found at: %s for writing", path);
2393      }
2394
2395      uint32_t bin_ctr = 0;
2396      while (bin_ctr < DAG_2_CACHE_SIZE) {
2397          struct dag_cache_entry_t *entry = cache[bin_ctr];
2398
2399          while (entry != NULL) {
2400              struct dag_t *dag = entry->dag;
2401              int ctr = 0;
2402              while (ctr < entry->instrs) {
2403                  fprintf(f, "%hi", entry->chain[ctr]);
2404                  if (ctr != entry->instrs - 1) {
2405                      fprintf(f, ", ");
2406                  }
2407                  ctr++;
2408              }
2409              fprintf(f, "\n");
2410              fprintf(f, "%u\n", dag->weight);
2411              fprintf(f, "%u\n", dag->max_roots);
2412
2413              char *dag_str = csp_dag_string(csound, dag);
2414              if (dag_str != NULL) {
2415                  fprintf(f, "%s", dag_str);
2416                  free(dag_str);
2417              }
2418              fprintf(f, "\n");
2419              entry = entry->next;
2420          }
2421
2422          bin_ctr++;
2423      }
2424
2425      fclose(f);
```

```
2426 }
2427
2428 static int csp_dag_cache_entry_alloc(CSOUND *csound,
2429                                      struct dag_cache_entry_t **entry,
2430                                      INSDS *chain,
2431                                      uint32_t hash_val)
2432 {
2433     if (entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
2434     if (chain == NULL) csound->Die(csound, "Invalid NULL Parameter chain");
2435
2436     int ctr = 0;
2437     INSDS *current_insds = chain;
2438     while (current_insds != NULL) {
2439         ctr++;
2440         current_insds = current_insds->nxtact;
2441     }
2442
2443     *entry = csound->Malloc(csound, sizeof(struct dag_cache_entry_t) +
2444                                         sizeof(int16) * ctr);
2445     if (*entry == NULL) {
2446         csound->Die(csound, "Failed to allocate Dag2 cache entry");
2447     }
2448     memset(*entry, 0, sizeof(struct dag_cache_entry_t) + sizeof(int16) * ctr);
2449     (*entry)->uses   = 1;
2450     (*entry)->age    = DAG_2_AGE_START;
2451     (*entry)->instrs = ctr;
2452     (*entry)->hash_val = hash_val;
2453
2454     ctr = 0;
2455     current_insds = chain;
2456     while (current_insds != NULL) {
2457         (*entry)->chain[ctr] = current_insds->insno;
2458         ctr++;
2459         current_insds = current_insds->nxtact;
2460     }
2461
2462     struct dag_t *dag = NULL;
2463     csp_dag_build(csound, &dag, chain);
2464     (*entry)->dag = dag;
2465     csp_dag_optimization(csound, dag);
2466
2467     return CSOUND_SUCCESS;
2468 }
2469
2470 static int csp_dag_cache_entry_dealloc(CSOUND *csound,
2471                                        struct dag_cache_entry_t **entry)
2472 {
2473     if (entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
2474     if (*entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
2475
2476     csp_dag_dealloc(csound, &((*entry)->dag));
2477
2478     csound->Free(csound, *entry);
2479     *entry = NULL;
2480
2481     return CSOUND_SUCCESS;
2482 }
2483
2484 static void csp_dag_cache_update(CSOUND *csound)
2485 {
2486     if (cache_ctr < DAG_2_CACHE_SIZE) {
2487         return;
```

```
2488      }
2489      csound->Message(csound, "Cache Update\n");
2490
2491      uint32_t bin_ctr = 0;
2492      while (bin_ctr < DAG_2_CACHE_SIZE) {
2493          if (cache[bin_ctr] == NULL) {
2494              bin_ctr++;
2495              continue;
2496          }
2497
2498          struct dag_cache_entry_t *entry = cache[bin_ctr], *prev = NULL;
2499          while (entry != NULL) {
2500              entry->uses = entry->uses >> DAG_2_DECAY_COMP;
2501              entry->age  = entry->age  >> DAG_2_DECAY_COMP;
2502              if (entry->uses < DAG_2_MIN_USE_LIMIT &&
2503                      entry->age < DAG_2_MIN_AGE_LIMIT) {
2504                  if (prev == NULL) {
2505                      cache[bin_ctr] = entry->next;
2506                  } else {
2507                      prev->next = entry->next;
2508                  }
2509                  csp_dag_cache_entry_dealloc(csound, &entry);
2510                  if (prev == NULL) {
2511                      entry = cache[bin_ctr];
2512                  } else {
2513                      entry = prev->next;
2514                  }
2515                  cache_ctr--;
2516              } else {
2517                  prev = entry;
2518                  entry = entry->next;
2519              }
2520          }
2521          bin_ctr++;
2522      }
2523  }
2524
2525  static int csp_dag_cache_compare(CSOUND *csound,
2526                                   struct dag_cache_entry_t *entry,
2527                                   INSDS *chain)
2528  {
2529      /* if (entry == NULL) csound->Die(csound, "Invalid NULL Parameter entry");
2530      if (chain == NULL) csound->Die(csound, "Invalid NULL Parameter chain"); */
2531
2532      INSDS *current_insds = chain;
2533      int32_t ctr = 0;
2534      while (current_insds != NULL && ctr < entry->instrs) {
2535          if (current_insds->insno != entry->chain[ctr]) {
2536              return 0;
2537          }
2538          current_insds = current_insds->nxtact;
2539          ctr++;
2540      }
2541      if (ctr >= entry->instrs && current_insds != NULL) {
2542          return 0;
2543      } else if (ctr < entry->instrs && current_insds == NULL) {
2544          return 0;
2545      } else {
2546          return 1;
2547      }
2548  }
2549
```

```
2550 void csp_dag_cache_fetch(CSOUND *csound, struct dag_t **dag, INSDS *chain)
2551 {
2552     /* if (dag == NULL) csound->Die(csound, "Invalid NULL Parameter dag");
2553     if (chain == NULL) csound->Die(csound, "Invalid NULL Parameter chain"); */
2554
2555     /* static int ctr = 0; */
2556
2557     update_ctr++;
2558     if (update_ctr == 10000) {
2559         csp_dag_cache_update(csound);
2560         update_ctr = 0;
2561
2562 #ifdef HYBRID_HASH_CACHE
2563         cache_last = NULL;
2564 #endif
2565     }
2566
2567 #ifdef HYBRID_HASH_CACHE
2568     if (cache_last != NULL && csp_dag_cache_compare(csound, cache_last, chain)) {
2569   #if TRACE > 2
2570         csound->Message(csound, "Cache Hit (Last) [%i]\n", cache_ctr);
2571   #endif
2572         struct dag_cache_entry_t *curr = cache_last;
2573
2574         *dag = curr->dag;
2575
2576         curr->uses++;
2577
2578         csp_dag_prepare_use_insds(csound, curr->dag, chain);
2579         csp_dag_prepare_use(csound, curr->dag);
2580         return;
2581     }
2582 #endif
2583
2584     uint32_t hash_val = hash_chain(chain, DAG_2_CACHE_SIZE);
2585     struct dag_cache_entry_t *curr = cache[hash_val];
2586     while (curr != NULL) {
2587         if (csp_dag_cache_compare(csound, curr, chain)) {
2588             TRACE_2("Cache Hit [%i]\n", cache_ctr);
2589
2590             *dag = curr->dag;
2591
2592             curr->uses++;
2593
2594             csp_dag_prepare_use_insds(csound, curr->dag, chain);
2595             csp_dag_prepare_use(csound, curr->dag);
2596 #ifdef HYBRID_HASH_CACHE
2597             cache_last = curr;
2598 #endif
2599             break;
2600         }
2601         curr = curr->next;
2602     }
2603     if (*dag == NULL) {
2604         TRACE_2("Cache Miss [%i]\n", cache_ctr);
2605         csp_dag_cache_entry_alloc(csound, &curr, chain, hash_val);
2606         cache_ctr++;
2607         *dag = curr->dag;
2608
2609         curr->next = cache[hash_val];
2610         cache[hash_val] = curr;
2611 #ifdef HYBRID_HASH_CACHE
```

```
2612         cache_last = curr;
2613 #endif
2614     }
2615 }
2616
2617 #endif
```

A.2.4 File: csound_orc.y

```
1  /*
2     csound_orc.y:
3
4     Copyright (C) 2006, 2007
5     John ffitch, Steven Yi
6
7     This file is part of Csound.
8
9     The Csound Library is free software; you can redistribute it
10    and/or modify it under the terms of the GNU Lesser General Public
11    License as published by the Free Software Foundation; either
12    version 2.1 of the License, or (at your option) any later version.
13
14    Csound is distributed in the hope that it will be useful,
15    but WITHOUT ANY WARRANTY; without even the implied warranty of
16    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17    GNU Lesser General Public License for more details.
18
19    You should have received a copy of the GNU Lesser General Public
20    License along with Csound; if not, write to the Free Software
21    Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
22    02111-1307 USA
23 */
24 %token S_COM
25 %token S_Q
26 %token S_COL
27 %token S_NOT
28 %token S_PLUS
29 %token S_MINUS
30 %token S_TIMES
31 %token S_DIV
32 %token S_NL
33 %token S_LB
34 %token S_RB
35 %token S_NEQ
36 %token S_AND
37 %token S_OR
38 %token S_LT
39 %token S_LE
40 %token S_EQ
41 %token S_ASSIGN
42 %token S_GT
43 %token S_GE
44 %token S_XOR
45 %token S_MOD
46
47 %token T_LABEL
48 %token T_IF
49
50 %token T_OPCODE0
51 %token T_OPCODE
52
```

```
53 %token T_UDO
54 %token T_UDOSTART
55 %token T_UDOEND
56 %token T_UDO_ANS
57 %token T_UDO_ARGS
58
59 %token T_ERROR
60
61 %token T_FUNCTION
62
63 %token T_INSTR
64 %token T_ENDIN
65 %token T_STRSET
66 %token T_PSET
67 %token T_CTRLINIT
68 %token T_MASSIGN
69 %token T_TURNON
70 %token T_PREALLOC
71 %token T_ZAKINIT
72 %token T_FTGEN
73 %token T_INIT
74 %token T_GOTO
75 %token T_KGOTO
76 %token T_IGOTO
77
78 %token T_SRATE
79 %token T_KRATE
80 %token T_KSMPS
81 %token T_NCHNLS
82 %token T_STRCONST
83 %token T_IDENT
84
85 %token T_IDENT_I
86 %token T_IDENT_GI
87 %token T_IDENT_K
88 %token T_IDENT_GK
89 %token T_IDENT_A
90 %token T_IDENT_GA
91 %token T_IDENT_W
92 %token T_IDENT_GW
93 %token T_IDENT_F
94 %token T_IDENT_GF
95 %token T_IDENT_S
96 %token T_IDENT_GS
97 %token T_IDENT_P
98 %token T_IDENT_B
99 %token T_IDENT_b
100 %token T_INTGR
101 %token T_NUMBER
102 %token T_THEN
103 %token T_ITHEN
104 %token T_KTHEN
105 %token T_ELSEIF
106 %token T_ELSE
107 %token T_ENDIF
108
109 %start orcfile
110 %left S_AND S_OR
111 %nonassoc S_LT S_GT S_LEQ S_GEQ S_EQ S_NEQ
112 %nonassoc T_THEN T_ITHEN T_KTHEN T_ELSE /* NOT SURE IF THIS IS NECESSARY */
113 %left S_PLUS S_MINUS
114 %left S_STAR S_SLASH
```

```
115 %right S_UNOT
116 %right S_UMINUS
117 %token S_GOTO
118 %token T_HIGHEST
119 %pure_parser
120 %error-verbose
121 %parse-param { CSOUND * csound }
122 %parse-param { TREE * astTree }
123 %lex-param { CSOUND * csound }
124
125 /* NOTE: Perhaps should use %union feature of bison */
126
127 %{
128 /* #define YYSTYPE ORCTOKEN* */
129 /* JPff thinks that line must be wrong and is trying this! */
130 #define YYSTYPE TREE*
131
132 #ifndef NULL
133 #define NULL 0L
134 #endif
135 #include "csoundCore.h"
136 #include <ctype.h>
137 #include "namedins.h"
138
139 #include "csound_orc.h"
140
141 #include "cs_par_base.h"
142 #include "cs_par_orc_semantic_analysis.h"
143
144 int udoflag = -1; /* THIS NEEDS TO BE MADE NON-GLOBAL */
145 int namedInstrFlag = 0; /* THIS NEEDS TO BE MADE NON-GLOBAL */
146
147 extern TREE* appendToTree(CSOUND * csound, TREE *first, TREE *newlast);
148 extern int csound_orclex (TREE*, CSOUND *);
149 extern void print_tree(CSOUND *, TREE *);
150 extern void csound_orcerror(CSOUND *, TREE*, char*);
151 extern void add_udo_definition(CSOUND*, char *, char *, char *);
152 %}
153 %%
154
155 orcfile           : rootstatement
156                     {
157                         *astTree = *((TREE *)$1);
158                         csp_orc_sa_print_list(csound);
159                     }
160                   ;
161
162 rootstatement     : rootstatement topstatement
163                     {
164                     $$ = appendToTree(csound, $1, $2);
165                     }
166                   | rootstatement instrdecl
167                     {
168                     $$ = appendToTree(csound, $1, $2);
169                     }
170                   | rootstatement udodecl
171                     {
172                     $$ = appendToTree(csound, $1, $2);
173                     }
174                   | topstatement
175                   | instrdecl
176                   | udodecl
```

```
177                     ;
178
179 /* FIXME: Does not allow "instr 2,3,4,5,6" syntax */
180 /* FIXME: Does not allow named instruments i.e. "instr trumpet" */
181 instrdecl : T_INSTR
182                 { namedInstrFlag = 1; }
183               T_INTGR S_NL
184                 {
185                     namedInstrFlag = 0;
186                     csp_orc_sa_instr_add(csound, ((ORCTOKEN *)$3)->lexeme);
187                 }
188               statementlist T_ENDIN S_NL
189                 {
190                     TREE *leaf = make_leaf(csound, T_INTGR, (ORCTOKEN *)$3);
191                     $$ = make_node(csound, T_INSTR, leaf, $6);
192                     csp_orc_sa_instr_finalize(csound);
193                 }
194
195           | T_INSTR
196                 { namedInstrFlag = 1; }
197               T_IDENT S_NL
198                 {
199                     namedInstrFlag = 0;
200                     csp_orc_sa_instr_add(csound, ((ORCTOKEN *)$3)->lexeme);
201                 }
202               statementlist T_ENDIN S_NL
203                 {
204                     TREE *ident = make_leaf(csound, T_IDENT, (ORCTOKEN *)$3);
205                     $$ = make_node(csound, T_INSTR, ident, $6);
206                     csp_orc_sa_instr_finalize(csound);
207                 }
208
209           | T_INSTR S_NL error
210                 {
211                     namedInstrFlag = 0;
212                     csound->Message(csound, Str("No number following instr\n"));
213                     csp_orc_sa_instr_finalize(csound);
214                 }
215           ;
216
217 udodecl   : T_UDOSTART
218                                                 { udoflag = -2; }
219                   T_IDENT
220                                                 { udoflag = -1; }
221                   S_COM
222                                                 { udoflag = 0;}
223             T_UDO_ANS
224                         { udoflag = 1; }
225             S_COM T_UDO_ARGS S_NL
226             {
227               udoflag = 2;
228               add_udo_definition(csound,
229                       ((ORCTOKEN *)$3)->lexeme,
230                       ((ORCTOKEN *)$7)->lexeme,
231                       ((ORCTOKEN *)$10)->lexeme);
232             }
233             statementlist T_UDOEND S_NL
234             {
235               udoflag = -1;
236
237
238               csound->Message(csound, "UDO COMPLETE\n");
```

```
239                 TREE *udoTop = make_leaf(csound, T_UDO, (ORCTOKEN *)NULL);
240                 TREE *ident = make_leaf(csound, T_IDENT, (ORCTOKEN *)$3);
241                 TREE *udoAns = make_leaf(csound, T_UDO_ANS, (ORCTOKEN *)$7);
242                 TREE *udoArgs = make_leaf(csound, T_UDO_ARGS, (ORCTOKEN *)$10);
243
244                 udoTop->left = ident;
245                 ident->left = udoAns;
246                 ident->right = udoArgs;
247
248                 udoTop->right = (TREE *)$13;
249
250                 $$ = udoTop;
251
252                 print_tree(csound, (TREE *)$$);
253
254               }
255
256             ;
257
258
259 statementlist : statementlist statement
260                 {
261                     $$ = appendToTree(csound, (TREE *)$1, (TREE *)$2);
262                 }
263                 | /* null */          { $$ = NULL; }
264                 ;
265
266 topstatement : rident S_ASSIGN expr S_NL
267                 {
268
269                   TREE *ans = make_leaf(csound, S_ASSIGN, (ORCTOKEN *)$2);
270                   ans->left = (TREE *)$1;
271                   ans->right = (TREE *)$3;
272                   /* ans->value->lexeme = get_assignment_type(csound,
273                       ans->left->value->lexeme, ans->right->value->lexeme); */
274
275                   $$ = ans;
276                 }
277                 | statement { $$ = $1; }
278
279              ;
280
281 statement : ident S_ASSIGN expr S_NL
282                 {
283
284                     TREE *ans = make_leaf(csound, S_ASSIGN, (ORCTOKEN *)$2);
285                     ans->left = (TREE *)$1;
286                     ans->right = (TREE *)$3;
287                     /* ans->value->lexeme = get_assignment_type(csound,
288                        ans->left->value->lexeme, ans->right->value->lexeme); */
289
290                     $$ = ans;
291
292                     csp_orc_sa_global_read_write_add_list(csound,
293                                         csp_orc_sa_globals_find(csound, ans->left),
294                                         csp_orc_sa_globals_find(csound, ans->right));
295                 }
296           | ans opcode exprlist S_NL
297                 {
298
299                     $2->left = $1;
300                     $2->right = $3;
```

```
301
302                     $$ = $2;
303
304                     csp_orc_sa_global_read_write_add_list(csound,
305                                     csp_orc_sa_globals_find(csound, $2->left),
306                                     csp_orc_sa_globals_find(csound, $2->right));
307                 }
308             | opcode0 exprlist S_NL
309                 {
310                     ((TREE *)$1)->left = NULL;
311                     ((TREE *)$1)->right = (TREE *)$2;
312
313                     $$ = $1;
314
315                     csp_orc_sa_global_read_add_list(csound,
316                                     csp_orc_sa_globals_find(csound, $1->right));
317                 }
318             | T_LABEL S_NL
319                 {
320                     $$ = make_leaf(csound, T_LABEL, (ORCTOKEN *)yylval);
321                 }
322             | goto T_IDENT S_NL
323                 {
324                     $1->left = NULL;
325                     $1->right = make_leaf(csound, T_IDENT, (ORCTOKEN *)$2);
326                     $$ = $1;
327                 }
328             | T_IF S_LB expr S_RB goto T_IDENT S_NL
329                 {
330                     $5->left = NULL;
331                     $5->right = make_leaf(csound, T_IDENT, (ORCTOKEN *)$6);
332                     $$ = make_node(csound, T_IF, $3, $5);
333                 }
334
335             | ifthen
336             | S_NL { $$ = NULL; }
337             ;
338
339 ans         : ident                 { $$ = $1; }
340             | ans S_COM ident       { $$ = appendToTree(csound, $1, $3); }
341             ;
342
343 ifthen      : T_IF S_LB expr S_RB then S_NL statementlist T_ENDIF S_NL
344             {
345               $5->right = $7;
346               $$ = make_node(csound, T_IF, $3, $5);
347             }
348             | T_IF S_LB expr S_RB then S_NL statementlist T_ELSE statementlist T_ENDIF S_NL
349             {
350               $5->right = $7;
351               $5->next = make_node(csound, T_ELSE, NULL, $9);
352               $$ = make_node(csound, T_IF, $3, $5);
353
354             }
355             | T_IF S_LB expr S_RB then S_NL statementlist elseiflist T_ENDIF S_NL
356             {
357               csound->Message(csound, "IF-ELSEIF FOUND!\n");
358               $5->right = $7;
359               $5->next = $8;
360               $$ = make_node(csound, T_IF, $3, $5);
361             }
362             | T_IF S_LB expr S_RB then S_NL statementlist elseiflist T_ELSE
```

```
363           statementlist T_ENDIF S_NL
364           {
365             csound->Message(csound, "IF-ELSEIF-ELSE FOUND!\n");
366             TREE * tempLastNode;
367
368             $5->right = $7;
369             $5->next = $8;
370
371             $$ = make_node(csound, T_IF, $3, $5);
372
373             tempLastNode = $$;
374
375             while(tempLastNode->right != NULL && tempLastNode->right->next != NULL) {
376                 tempLastNode = tempLastNode->right->next;
377             }
378
379             tempLastNode->right->next = make_node(csound, T_ELSE, NULL, $10);
380
381           }
382           ;
383
384 elseiflist : elseiflist elseif
385             {
386                 TREE * tempLastNode = $1;
387
388                 while(tempLastNode->right != NULL && tempLastNode->right->next != NULL) {
389                     tempLastNode = tempLastNode->right->next;
390                 }
391
392                 tempLastNode->right->next = $2;
393                 $$ = $1;
394             }
395             | elseif { $$ = $1; }
396            ;
397
398 elseif     : T_ELSEIF S_LB expr S_RB then S_NL statementlist
399             {
400                 csound->Message(csound, "ELSEIF FOUND!\n");
401                 $5->right = $7;
402                 $$ = make_node(csound, T_ELSEIF, $3, $5);
403             }
404           ;
405
406 then       : T_THEN
407             { $$ = make_leaf(csound, T_THEN, (ORCTOKEN *)yylval); }
408           | T_KTHEN
409             { $$ = make_leaf(csound, T_KTHEN, (ORCTOKEN *)yylval); }
410           | T_ITHEN
411             { $$ = make_leaf(csound, T_ITHEN, (ORCTOKEN *)yylval); }
412           ;
413
414
415 goto   : T_GOTO
416             { $$ = make_leaf(csound, T_GOTO, (ORCTOKEN *)yylval); }
417           | T_KGOTO
418             { $$ = make_leaf(csound, T_KGOTO, (ORCTOKEN *)yylval); }
419           | T_IGOTO
420             { $$ = make_leaf(csound, T_IGOTO, (ORCTOKEN *)yylval); }
421           ;
422
423
424 exprlist   : exprlist S_COM expr
```

```
425                 {
426                     /* $$ = make_node(S_COM, $1, $3); */
427                     $$ = appendToTree(csound, $1, $3);
428                 }
429           | exprlist S_COM error
430           | expr { $$ = $1;     }
431           | /* null */          { $$ = NULL; }
432           ;
433
434 expr      : expr S_Q expr S_COL expr %prec S_Q
435             { $$ = make_node(csound, S_Q, $1,
436                              make_node(csound, S_COL, $3, $5)); }
437           | expr S_Q expr S_COL error
438           | expr S_Q expr error
439           | expr S_Q error
440           | expr S_LE expr      { $$ = make_node(csound, S_LE, $1, $3); }
441           | expr S_LE error
442           | expr S_GE expr      { $$ = make_node(csound, S_GE, $1, $3); }
443           | expr S_GE error
444           | expr S_NEQ expr     { $$ = make_node(csound, S_NEQ, $1, $3); }
445           | expr S_NEQ error
446           | expr S_EQ expr      { $$ = make_node(csound, S_EQ, $1, $3); }
447           | expr S_EQ error
448           | expr S_GT expr      { $$ = make_node(csound, S_GT, $1, $3); }
449           | expr S_GT error
450           | expr S_LT expr      { $$ = make_node(csound, S_LT, $1, $3); }
451           | expr S_LT error
452           | expr S_AND expr     { $$ = make_node(csound, S_AND, $1, $3); }
453           | expr S_AND error
454           | expr S_OR expr      { $$ = make_node(csound, S_OR, $1, $3); }
455           | expr S_OR error
456           | S_NOT expr %prec S_UNOT { $$ = make_node(csound, S_UNOT, $2, NULL); }
457           | S_NOT error
458           | iexp                { $$ = $1; }
459           ;
460
461 iexp      : iexp S_PLUS iterm   { $$ = make_node(csound, S_PLUS, $1, $3); }
462           | iexp S_PLUS error
463           | iexp S_MINUS iterm  { $$ = make_node(csound, S_MINUS, $1, $3); }
464           | expr S_MINUS error
465           | iterm               { $$ = $1; }
466           ;
467
468 iterm     : iterm S_TIMES ifac  { $$ = make_node(csound, S_TIMES, $1, $3); }
469           | iterm S_TIMES error
470           | iterm S_DIV ifac    { $$ = make_node(csound, S_DIV, $1, $3); }
471           | iterm S_DIV error
472           | ifac                { $$ = $1; }
473           ;
474
475 ifac      : ident               { $$ = $1; }
476           | constant            { $$ = $1; }
477           | S_MINUS ifac %prec S_UMINUS
478             {
479                 $$ = make_node(csound, S_UMINUS, NULL, $2);
480             }
481           | S_MINUS error
482           | S_LB expr S_RB      { $$ = $2; }
483           | S_LB expr error
484           | S_LB error
485           | function S_LB exprlist S_RB
486             {
```

```
487                 $1->left = NULL;
488                 $1->right = $3;
489
490                 $$ = $1;
491               }
492             | function S_LB error
493             ;
494
495 function    : T_FUNCTION  { $$ = make_leaf(csound, T_FUNCTION, (ORCTOKEN *)$1); }
496             ;
497
498 /* exprstrlist     : exprstrlist S_COM expr
499                                          { $$ = make_node(csound, S_COM, $1, $3); }
500             | exprstrlist S_COM T_STRCONST
501                   { $$ = make_node(csound, S_COM, $1,
502                     make_leaf(csound, T_STRCONST, (ORCTOKEN *)yylval)); }
503             | exprstrlist S_COM error
504             | expr                { $$ = $1; }
505             ;
506  */
507
508 rident      : T_SRATE     { $$ = make_leaf(csound, T_SRATE, (ORCTOKEN *)yylval); }
509             | T_KRATE     { $$ = make_leaf(csound, T_KRATE, (ORCTOKEN *)yylval); }
510             | T_KSMPS     { $$ = make_leaf(csound, T_KSMPS, (ORCTOKEN *)yylval); }
511             | T_NCHNLS    { $$ = make_leaf(csound, T_NCHNLS, (ORCTOKEN *)yylval); }
512             ;
513
514 ident       : T_IDENT_I   { $$ = make_leaf(csound, T_IDENT_I, (ORCTOKEN *)yylval); }
515             | T_IDENT_K   { $$ = make_leaf(csound, T_IDENT_K, (ORCTOKEN *)yylval); }
516             | T_IDENT_F   { $$ = make_leaf(csound, T_IDENT_F, (ORCTOKEN *)yylval); }
517             | T_IDENT_W   { $$ = make_leaf(csound, T_IDENT_W, (ORCTOKEN *)yylval); }
518             | T_IDENT_S   { $$ = make_leaf(csound, T_IDENT_S, (ORCTOKEN *)yylval); }
519             | T_IDENT_A   { $$ = make_leaf(csound, T_IDENT_A, (ORCTOKEN *)yylval); }
520             | T_IDENT_P   { $$ = make_leaf(csound, T_IDENT_P, (ORCTOKEN *)yylval); }
521             | gident      { $$ = $1; }
522             ;
523
524 gident      : T_IDENT_GI  { $$ = make_leaf(csound, T_IDENT_GI, (ORCTOKEN *)yylval); }
525             | T_IDENT_GK  { $$ = make_leaf(csound, T_IDENT_GK, (ORCTOKEN *)yylval); }
526             | T_IDENT_GF  { $$ = make_leaf(csound, T_IDENT_GF, (ORCTOKEN *)yylval); }
527             | T_IDENT_GW  { $$ = make_leaf(csound, T_IDENT_GW, (ORCTOKEN *)yylval); }
528             | T_IDENT_GS  { $$ = make_leaf(csound, T_IDENT_GS, (ORCTOKEN *)yylval); }
529             | T_IDENT_GA  { $$ = make_leaf(csound, T_IDENT_GA, (ORCTOKEN *)yylval); }
530             ;
531
532 constant    : T_INTGR     { $$ = make_leaf(csound, T_INTGR, (ORCTOKEN *)yylval); }
533             | T_NUMBER    { $$ = make_leaf(csound, T_NUMBER, (ORCTOKEN *)yylval); }
534             | T_STRCONST  { $$ = make_leaf(csound, T_STRCONST, (ORCTOKEN *)yylval); }
535             | T_SRATE     { $$ = make_leaf(csound, T_NUMBER, (ORCTOKEN *)yylval); }
536             | T_KRATE     { $$ = make_leaf(csound, T_NUMBER, (ORCTOKEN *)yylval); }
537             | T_KSMPS     { $$ = make_leaf(csound, T_NUMBER, (ORCTOKEN *)yylval); }
538             | T_NCHNLS    { $$ = make_leaf(csound, T_NUMBER, (ORCTOKEN *)yylval); }
539             ;
540
541 opcode0     : T_OPCODE0
542               {
543                   csound->Message(csound, "opcode0 yylval=%p\n", yylval);
544                   $$ = make_leaf(csound, T_OPCODE0, (ORCTOKEN *)yylval);
545               }
546             ;
547
548 opcode      : T_OPCODE    { $$ = make_leaf(csound, T_OPCODE, (ORCTOKEN *)yylval); }
```

```
549             ;
550
551 %%
```

A.3 Source Snippets

All of the snippets in this section are modifications to existing files in Csound. We present them and describe their context.

A.3.1 File: entry1.h

```
1 int       globallock(CSOUND *, void *);
2 int       globalunlock(CSOUND *, void *);
```

A.3.2 File: entry1.c

Two entries into the table of opcodes.

```
1 { "globallock",   S(GLOBAL_LOCK_UNLOCK), 3, "", "k", globallock,   globallock,   NULL},
2 { "globalunlock", S(GLOBAL_LOCK_UNLOCK), 3, "", "k", globalunlock, globalunlock, NULL},
```

A.3.3 File: aops.h

```
1 typedef struct {
2     OPDS    h;
3     MYFLT   *gvar_ix;
4 } GLOBAL_LOCK_UNLOCK;
```

A.3.4 File: aops.c

```
1  int globallock(CSOUND *csound, GLOBAL_LOCK_UNLOCK *p)
2  {
3      /* csound->Message(csound, "Locking:   %i\n", (int)*p->gvar_ix); */
4      csp_locks_lock(csound, (int)*p->gvar_ix);
5      return OK;
6  }
7
8  int globalunlock(CSOUND *csound, GLOBAL_LOCK_UNLOCK *p)
9  {
10     /* csound->Message(csound, "UnLocking: %i\n", (int)*p->gvar_ix); */
11     csp_locks_unlock(csound, (int)*p->gvar_ix);
12     return OK;
13 }
```

A.3.5 File: csoundCore.h

Entries into the CSOUND structure (called struct CSOUND_ in csoundCore.h)

```
1 into CSOUND struct (struct CSOUND_ in csoundCore
2 char                  *weight_info;
3 char                  *weight_dump;
```

```
4 char                  *weights;
5 struct dag_t         *multiThreadedDag;
6 struct barrier_spin_t *barrier1;
7 struct barrier_spin_t *barrier2;
```

A.3.6 File: SConstruct

This file is the instructions for the build system. We added the necessary extra files to the newParserSources list.

```
1 newParserSources = Split('''
2 Engine/csound_orclex.c
3 Engine/csound_orcparse.c
4 Engine/csound_orc_semantics.c
5 Engine/csound_orc_expressions.c
6 Engine/csound_orc_optimize.c
7 Engine/csound_orc_compile.c
8 Engine/cs_par_base.c
9 Engine/cs_par_orc_semantic_analysis.c
10 Engine/cs_par_dispatch.c
11 Engine/new_orc_parser.c
12 Engine/symbtab.c
13 ''')
```

A.3.7 File: argdecode.c

We added some extra arguments for the weights. They are located near the end of the static int decode_long(CSOUND *csound, char *s, int argc, char **argv) function.

```
1 else if (!(strncmp(s, "weight-info=", 12))) {
2   s += 12;
3   if (*s=='\0') dieu(csound, Str("no weight-info"));
4   csound->weight_info = s;
5   return 1;
6 }
7 else if (!(strncmp(s, "weight-dump=", 12))) {
8   s += 12;
9   if (*s=='\0') dieu(csound, Str("no weight-dump"));
10  csound->weight_dump = s;
11  return 1;
12 }
13 else if (!(strncmp(s, "weights=", 8))) {
14  s += 8;
15  if (*s=='\0') dieu(csound, Str("no weights"));
16  csound->weights = s;
17  return 1;
18 }
19 else if (!(strcmp(s, "compute-weights"))) {
20  O->calculateWeights = 1;
21  return 1;
22 }
```

A.3.8 File: main.c

At the top of the file we have the necessary includes.

```
#include "cs_par_base.h"
#include "cs_par_orc_semantic_analysis.h"
#include "cs_par_dispatch.h"
```

At the end of the function PUBLIC int csoundCompile(CSOUND *csound, int argc, char **argv) we have the following to set up the barriers and start the worker threads.

```
    if (O->numThreads > 1) {
        int i;
        THREADINFO *current = NULL;

#ifdef NUM_THREADS_OLD_DEF
        csound->multiThreadedBarrier1 = csound->CreateBarrier(O->numThreads + 1);
        csound->multiThreadedBarrier2 = csound->CreateBarrier(O->numThreads + 1);
#else
        csound->multiThreadedBarrier1 = csound->CreateBarrier(O->numThreads/* + 1*/);
        csound->multiThreadedBarrier2 = csound->CreateBarrier(O->numThreads/* + 1*/);
#endif

#if defined(SPINLOCK_BARRIER) || defined(SPINLOCK_2_BARRIER)
  #ifdef NUM_THREADS_OLD_DEF
        csp_barrier_alloc(csound, &(csound->barrier1), O->numThreads + 1);
        csp_barrier_alloc(csound, &(csound->barrier2), O->numThreads + 1);
  #else
        csp_barrier_alloc(csound, &(csound->barrier1), O->numThreads);
        csp_barrier_alloc(csound, &(csound->barrier2), O->numThreads);
  #endif
#endif

        csound->multiThreadedComplete = 0;

#ifdef NUM_THREADS_OLD_DEF
        for(i = 0; i < O->numThreads; i++) {
#else
        for(i = 1; i < O->numThreads; i++) {
#endif
            THREADINFO *t = csound->Malloc(csound, sizeof(THREADINFO));

            t->threadId = csound->CreateThread(&kperfThread, (void *)csound);
            t->next = NULL;

            if(current == NULL) {
                csound->multiThreadedThreadInfo = t;
            } else {
                current->next = t;
            }
            current = t;
        }

#if defined(SPINLOCK_BARRIER) || defined(SPINLOCK_2_BARRIER)
        csp_barrier_wait(csound, csound->barrier2);
#else
        csound->WaitBarrier(csound->multiThreadedBarrier2);
#endif

        csp_parallel_compute_spec_setup(csound);
    }
```

A.3.9 File: csound.c

Again we have the following headers.

```
#include "cs_par_base.h"
#include "cs_par_orc_semantic_analysis.h"
#include "cs_par_dispatch.h"
```

nodePerf is entirely new. kperf, kperfThread have extensive changes. csoundPerform has very little changes.

```
int inline nodePerf(CSOUND *csound, int index)
{
  struct instr_semantics_t *instr = NULL;
  INSDS *insds = NULL;
  OPDS  *opstart = NULL;
  int update_hdl = -1;
  int played_count = 0;
  struct dag_node_t *node;

  do {
    csp_dag_consume(csound, csound->multiThreadedDag, &node, &update_hdl);

    if (UNLIKELY(node == NULL)) {
      return played_count;
    }

    if (node->hdr.type == DAG_NODE_INDV) {
      instr = node->instr;
      insds = node->insds;
      played_count++;

      TRACE_2("[%i] Playing: %s [%p]\n", index, instr->name, insds);

      opstart = (OPDS *)insds;
      while ((opstart = opstart->nxtp) != NULL) {
        (*opstart->opadr)(csound, opstart); /* run each opcode */
      }

      TRACE_2("[%i] Played:  %s [%p]\n", index, instr->name, insds);
    } else if (node->hdr.type == DAG_NODE_LIST) {
      played_count += node->count;

      int node_ctr = 0;
      while (node_ctr < node->count) {
        struct dag_node_t *play_node = node->nodes[node_ctr];
        instr = play_node->instr;
        insds = play_node->insds;

        TRACE_2("[%i] Playing: %s [%p]\n", index, instr->name, insds);

        opstart = (OPDS *)insds;
        while ((opstart = opstart->nxtp) != NULL) {
          (*opstart->opadr)(csound, opstart); /* run each opcode */
        }

        TRACE_2("[%i] Played:  %s [%p]\n", index, instr->name, insds);
        node_ctr++;
      }
    } else if (node->hdr.type == DAG_NODE_DAG) {
      csound->Die(csound, "Recursive DAGs not implemented");
```

```
51        } else {
52          csound->Die(csound, "Unknown DAG node type");
53        }
54
55        csp_dag_consume_update(csound, csound->multiThreadedDag, update_hdl);
56      } while (!csp_dag_is_finished(csound, csound->multiThreadedDag));
57
58      return played_count;
59    }
60
61    unsigned long kperfThread(void * cs)
62    {
63        INSDS *start;
64        CSOUND *csound = (CSOUND *)cs;
65        void   *barrier1 = csound->multiThreadedBarrier1;
66        void   *barrier2 = csound->multiThreadedBarrier2;
67
68  #if defined(SPINLOCK_BARRIER) || defined(SPINLOCK_2_BARRIER)
69        csp_barrier_wait(csound, csound->barrier2);
70  #else
71        csound->WaitBarrier(barrier2);
72  #endif
73
74        void *threadId = csound->GetCurrentThreadID();
75        int index = getThreadIndex(csound, threadId);
76        int numThreads = csound->oparms->numThreads;
77        start = NULL;
78        csound->Message(csound,
79                        "Multithread performance: insno: %3d  thread %d of "
80                        "%d starting.\n",
81                        start ? start->insno : -1,
82                        index,
83                        numThreads);
84        if(index < 0) {
85          csound->Die(csound, "Bad ThreadId");
86          return ULONG_MAX;
87        }
88        index++;
89
90        while (1) {
91
92          TRACE_1("[%i] Barrier1 Reached\n", index);
93          SHARK_SIGNPOST(BARRIER_1_WAIT_SYM);
94  #if   defined(SPINLOCK_BARRIER) || defined(SPINLOCK_2_BARRIER)
95          csp_barrier_wait(csound, csound->barrier1);
96  #else
97          csound->WaitBarrier(barrier1);
98  #endif
99
100         TRACE_1("[%i] Go\n", index);
101
102         /* TIMER_INIT(mutex, "Mutex ")
103         TIMER_T_START(mutex, index, "Mutex ") */
104
105         csound_global_mutex_lock();
106         if (csound->multiThreadedComplete == 1) {
107           csound_global_mutex_unlock();
108           free(threadId);
109           /* csound->Message(csound,
110                           "Multithread performance: insno: %3d  thread "
111                           "%d of %d exiting.\n",
112                           start->insno,
```

```
113                             index,
114                             numThreads); */
115           return 0UL;
116         }
117         csound_global_mutex_unlock();
118
119         /* TIMER_T_END(mutex, index, "Mutex ") */
120
121         TIMER_INIT(thread, "")
122         TIMER_T_START(thread, index, "")
123
124 #if 0
125         if (start) {
126           end = csound->multiThreadedEnd;
127           numActive = getNumActive(start, end);
128           partitionWork(csound, &start, &end, index, numThreads, numActive);
129           csound->DebugMsg(csound,
130                            "kperfThread:       insno: %3d  thread: %3d  of: %3d"
131                            "  start: 0x%p  end: 0x%p  active: %3d\n",
132                            start->insno,
133                            index,
134                            numThreads,
135                            start,
136                            end,
137                            numActive);
138           while(start != NULL && start != end) {
139             opstart = (OPDS *)start;
140             while ((opstart = opstart->nxtp) != NULL) {
141               (*opstart->opadr)(csound, opstart); /* run each opcode */
142             }
143             start = start->nxtact;          /* ip = nxt; but that does not allow for
144                                                deletions */
145           }
146         }
147 #endif
148
149         nodePerf(csound, index);
150
151         TIMER_T_END(thread, index, "")
152
153         TRACE_1("[%i] Done\n", index);
154
155         SHARK_SIGNPOST(BARRIER_2_WAIT_SYM);
156 #if   defined(SPINLOCK_BARRIER) || defined(SPINLOCK_2_BARRIER)
157         csp_barrier_wait(csound, csound->barrier2);
158 #else
159         csound->WaitBarrier(barrier2);
160 #endif
161         TRACE_1("[%i] Barrier2 Done\n", index);
162       }
163   }
164
165   static inline int kperf(CSOUND *csound)
166   {
167 #ifndef OLPC
168       void *barrier1, *barrier2;
169 #endif
170       INSDS *ip;
171       /* update orchestra time */
172       csound->kcounter = ++(csound->global_kcounter);
173       csound->icurTime += csound->ksmps;
174       csound->curBeat += csound->curBeat_inc;
```

```
175       /* if skipping time on request by 'a' score statement: */
176       if (UNLIKELY(csound->advanceCnt)) {
177         csound->advanceCnt--;
178         return 1;
179       }
180       /* if i-time only, return now */
181       if (UNLIKELY(csound->initonly))
182         return 1;
183       /* PC GUI needs attention, but avoid excessively frequent */
184       /* calls of csoundYield() */
185       if (--(csound->evt_poll_cnt) < 0) {
186         csound->evt_poll_cnt = csound->evt_poll_maxcnt;
187         if (!csoundYield(csound))
188           csound->LongJmp(csound, 1);
189       }
190       /* for one kcnt: */
191       if (csound->oparms_.sfread)          /*   if audio_infile open  */
192         csound->spinrecv(csound);          /*      fill the spin buf  */
193       csound->spoutactive = 0;             /*   make spout inactive   */
194 #ifndef OLPC
195       barrier1 = csound->multiThreadedBarrier1;
196       barrier2 = csound->multiThreadedBarrier2;
197 #endif
198       ip = csound->actanchor.nxtact;
199
200       if (ip != NULL) {
201 #ifndef OLPC
202
203         TIMER_INIT(thread, "")
204         TIMER_START(thread, "Clock Sync ")
205         TIMER_END(thread, "Clock Sync ")
206
207         SHARK_SIGNPOST(KPERF_SYM);
208         TRACE_1("[%i] kperf\n", 0);
209
210         /* There are 2 partitions of work: 1st by inso,
211            2nd by inso count / thread count. */
212         if (csound->multiThreadedThreadInfo != NULL) {
213           struct dag_t *dag2 = NULL;
214           int main_played_count = 0;
215
216           TIMER_START(thread, "Dag ")
217   #if defined(LINEAR_CACHE) || defined(HASH_CACHE)
218           csp_dag_cache_fetch(csound, &dag2, ip);
219   #else
220           csp_dag_build(csound, &dag2, ip);
221   #endif
222           TIMER_END(thread, "Dag ")
223
224           TRACE_1("{Time: %f}\n", csound->GetScoreTime(csound));
225   #if TRACE > 1
226           csp_dag_print(csound, dag2);
227   #endif
228           csound->multiThreadedDag = dag2;
229
230           /* process this partition */
231           TRACE_1("[%i] Barrier1 Reached\n", 0);
232           SHARK_SIGNPOST(BARRIER_1_WAIT_SYM);
233   #if   defined(SPINLOCK_BARRIER) || defined(SPINLOCK_2_BARRIER)
234           csp_barrier_wait(csound, csound->barrier1);
235   #else
236           csound->WaitBarrier(barrier1);
```

```
237   #endif
238
239           TIMER_START(thread, "[0] ")
240
241   #ifndef NUM_THREADS_OLD_DEF
242           main_played_count = nodePerf(csound, 0);
243   #endif
244
245           TIMER_END(thread, "[0] ")
246
247           SHARK_SIGNPOST(BARRIER_2_WAIT_SYM);
248           /* wait until partition is complete */
249   #if    defined(SPINLOCK_BARRIER) || defined(SPINLOCK_2_BARRIER)
250           csp_barrier_wait(csound, csound->barrier2);
251   #else
252           csound->WaitBarrier(barrier2);
253   #endif
254           TRACE_1("[%i] Barrier2 Done\n", 0);
255           TIMER_END(thread, "")
256
257   #if !defined(LINEAR_CACHE) && !defined(HASH_CACHE)
258           csp_dag_dealloc(csound, &dag2);
259   #else
260           dag2 = NULL;
261   #endif
262           csound->multiThreadedDag = NULL;
263         }
264         else {
265 #endif
266           while (ip != NULL) {                /* for each instr active:  */
267             INSDS *nxt = ip->nxtact;
268             csound->pds = (OPDS*) ip;
269             while ((csound->pds = csound->pds->nxtp) != NULL) {
270               (*csound->pds->opadr)(csound, csound->pds); /* run each opcode */
271             }
272             ip = nxt; /* but this does not allow for all deletions */
273           }
274 #ifndef OLPC
275         }
276 #endif
277       }
278       if (!csound->spoutactive) {           /*   results now in spout? */
279         memset(csound->spout, 0, csound->nspout*sizeof(MYFLT));
280         /* csound->DebugMsg(csound,"Zero segment %d\n", csound->cyclesRemaining); */
281       }
282       csound->spoutran(csound);             /*      send to audio_out  */
283       return 0;
284   }
285
286   PUBLIC int csoundPerform(CSOUND *csound)
287   {
288       int done;
289       int returnValue;
290       csound->performState = 0;
291       /* setup jmp for return after an exit() */
292       if ((returnValue = setjmp(csound->exitjmp))) {
293 #ifndef MACOSX
294         csoundMessage(csound, "Early return from csoundPerform().\n");
295 #endif
296         return ((returnValue - CSOUND_EXITJMP_SUCCESS) | CSOUND_EXITJMP_SUCCESS);
297       }
298       do {
```

```
299         do {
300           if ((done = sensevents(csound))) {
301             csoundMessage(csound, "Score finished in csoundPerform().\n");
302
303             if (csound->oparms->numThreads > 1) {
304 #if    defined(LINEAR_CACHE) || defined(HASH_CACHE)
305               csp_dag_cache_print(csound);
306 #endif
307               csound->multiThreadedComplete = 1;
308
309 #if    defined(SPINLOCK_BARRIER) || defined(SPINLOCK_2_BARRIER)
310               csp_barrier_wait(csound, csound->barrier1);
311 #else
312               csound->WaitBarrier(barrier1);
313 #endif
314             }
315             if (csound->oparms->calculateWeights) {
316               /* csp_weights_dump(csound); */
317               csp_weights_dump_normalised(csound);
318             }
319
320             return done;
321           }
322         } while (kperf(csound));
323       } while ((unsigned char) csound->performState == (unsigned char) 0);
324       csoundMessage(csound, "csoundPerform(): stopped.\n");
325       csound->performState = 0;
326       return 0;
327   }
```